ESD-TR-80-217

(12)

LEVEL II

B.S.

AD A094064

# Technical Note 1980-50

Sec 1473

D. B. Paul
J. A. Feldman
V. J. Sferrino

A Design Study for an Easily Programmable, High-Speed Processor with a General-Purpose Architecture

23 October 1980

Prepared for the Defense Advanced Research Projects Agency under Electronic Systems Division Contract F19628-80-C-0002 by

**Lincoln Laboratory**

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

*LEXINGTON, MASSACHUSETTS*

81 1 23 004

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored by the Defense Advanced Research Projects Agency under Air Force Contract F19628-80-C-0002 (ARPA Order 3336).

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LINCOLN LABORATORY

# A DESIGN STUDY FOR AN EASILY PROGRAMMABLE, HIGH-SPEED PROCESSOR WITH A GENERAL-PURPOSE ARCHITECTURE

*D. B. PAUL*
*J. A. FELDMAN*
*V. J. SFERRINO*
*Group 24*

TECHNICAL NOTE 1980-50

23 OCTOBER 1980

LEXINGTON MASSACHUSETTS

## ABSTRACT

A design study has been carried out for an easy-to-program high-speed signal processor. The machine achieves a throughput of about 25 million arithmetic operations per second by incorporating parallel addressing and data access into a general-purpose architecture. The study indicates that, for typical signal processing applications, nearly all of the instruction cycles can be used to perform primary arithmetic operations rather than data access or addressing. The proposed machine would require about 1165 ECL 100K chips, occupy 2-3 cubic feet, and consume about 700 watts.

## CONTENTS

## I. INTRODUCTION

Toward the end of the 1960s the field of digital signal processing had become firmly established with many fundamental concepts and essential results already well known throughout the engineering and research and development communities. The digital circuit technology base of the era supported specialized hardware designs capable of reasonably high throughput at tractable levels of complexity. Thus for many applications of interest, the digital approach offered a legitimate, practical alternative to more classical signal processing techniques.

As more researchers become active in the field, it became increasingly clear that "standard" computational machinery was frustratingly inadequate as a general developmental tool. The amount of computation required in typical processing problems was enough to keep ordinary computers busy for hours producing perhaps only a few seconds of processed data. Therefore, many special-purpose hardware designs were frozen and committed to production based on a dangerously limited amount of performance simulation evidence - with frequently disastrous results. What was clearly needed was a flexible, programmable, computer-like processor which could support a substantially higher level of throughput than any conventional, general-purpose machine. The ideal situation, of course, would be to be able to emulate in real time the exact sequence of computation steps necessary in a given application and to operate on realistic raw data.

Lincoln Laboratory was among the first to confront this issue with the "Fast Digital Processor" (FDP)[1] which was designed and fabricated in the 1968-1970 time frame. This was a parallel, semi-pipelined design featuring four arithmetic elements, dual data memories, separate program memory, and a relatively wide instruction word. Based on 3.5 nsec emitter coupled logic technology, the FDP operated at a fundamental clock period of 150 nsec, could support a maximum throughput rate of 53 million fixed point operations per second, and required approximately 15,000 integrated circuits.

Several facts of life were learned from the FDP and other large, complex, explicitly parallel machines like it. Firstly, these machines were so intricate and cumbersome, they could never be produced or maintained economically on any kind of reasonable scale. Secondly, unless the structure of the problem at hand was just right, these machines tended to be very difficult to program. These problems were quickly recognized at Lincoln and in the commercial sector. For a particular specialized problem, a relatively dedicated, highly structured, parallel configuration might still be considered the best match. For example, a very successful single instruction/multiple data stream (SIMD) type of processor, called the "Parallel Microprogrammed Processor" (PMP),[2] was designed and fabricated at Lincoln for radar signal processing. But for a more general class of problems, another approach began to evolve.

Potential commercial suppliers of programmable signal processing machinery had to consider balancing a variety of factors in defining their products including hardware complexity, performance, ease of use, universality of appeal, reliability, maintainability and most importantly, cost. What emerged is a broad class of designs known generically and collectively as "array processors." Array processor designs vary widely in detail from one manufacturer to another, each representing the "optimal" set of tradeoffs in the eye of its conceiver. In today's marketplace, the potential customer can choose from such vendors as Computer Signal Processors, Signal Processing Systems, Floating Point Systems, Computer Design and Application, or Analogic.

However, all array processors share some basic points of commonality. Firstly, though any given design may feature some degree of pipelining and may or may not be fundamentally a single CPU structure, rarely if ever, are there explicitly parallel full data ALUs. This is simply too expensive a luxury. Secondly, array processors are most efficient when repetitive, well-defined calculations are to be performed on highly structured data sets. Clearly much signal processing of practical interest does fall into this category. Thirdly, array processors are usually intended to be operated as an adjunct to a general-purpose host facility rather than in a stand-alone mode. This arrangement can greatly simplify the programmer's interaction with the device if supported with a properly designed software operating system. Finally, all array processors, without exception, are extremely difficult and time consuming to program at the assembly language level. The conceptual architectures tend to be extremely complex, and the unwary user can be quickly overcome by a variety of pitfalls.

The commercial vendors have expended a great deal of time, energy, and money in decoupling the user from the programming difficulties characteristic of array processors by precoding a large number of commonly encountered signal processing functions. These are stored in a host-based library and can be called from a higher level language (such as FORTRAN) as subroutines at compilation time. This strategy works well for a large percentage of applications, but not necessarily all. For example, in speech processing it frequently happens that processing functions are required which are not likely to be found in a standard subroutine library. It also may happen that the precoded software is not as efficient in running time as might be desired. It is necessary in such cases to be able to develop software quickly and expeditiously at the assembly language level.

This final requirement has given rise to yet another class of programmable signal processor which has been vigorously (but not solely) pursued by Lincoln. The Lincoln "Digital Voice Terminal" (LDVT)[3] and "Digital Signal Processor" (LDSP),[4] developed in 1974 and 1977 respectively, are examples of this last philosophical category. The basic groundrule in this class of designs is to develop as much raw throughput as possible through the technology base (e.g., 2 nsec ECL) but yet maintain as simple an architectural structure as possible to enhance programmability. Both the LDVT and LDSP feature very straightforward, single CPU, minimally pipelined, Von Neuman-type architectures and can be programmed as conveniently as second-generation mini-computers at the assembly language level.

These machines feature 20 MIPS (million instructions per second) throughput capabilities, are of very modest hardware complexity, and have generally been considered highly successful in meeting desired objectives in the context of the Lincoln speech processing programs.

With the relatively recent emergence of newer, subnanosecond digital circuit technologies and problems of ever-increasing complexity being frequently encountered in the speech research field, it is appropriate and timely to look beyond the LDSP to a potential successor. The fundamental objective of the study summarized in this report was to define a signal processing-oriented computer similar in philosophical approach to the LDVT/LDSP family, but on the order of 5 times as powerful for typical processing functions. The virtues of easy programming and moderate hardware complexity were to be stressed.

Experience with the LDSP has motivated the inclusion of several additional features. The 16 bit (integer) data word has been found to be inadequate in a number of applications, thereby suggesting a 24 bit data word. Users have also indicated a requirement for a floating point format which could be accommodated in the proposed word length. The sizes of both the program (2K) and data (4K) memories have been found to be a limitation in several applications. Therefore, the sizes of the memories are doubled in this design. Since the I/O capabilities of the LDSP have been judged adequate in the speech research facility environment, the same basic scheme (i.e., eight fully buffered, duplex vector interrupt I/O ports) will be assumed in the sequel.

These goals may be achieved through a combination of mechanisms. The ECL 100K logic family now commercially available, is about 2.5 times faster than the ECL 10K series (.75 ns versus 2 ns gate delay)[5] used to implement the LDSP. In practice, physical realities such as finite signal propagation delays over wires will prevent the theoretical speed-up factor from being achieved. An architecture supporting parallel indexing and a three-address arithmetic instruction format will greatly reduce addressing and data accessing overhead which constitutes as much as two thirds of the time of some loops programmed on the LDSP. The two-cycle multiply instruction of the LDSP can be combined with an add in two cycles to create an effective one-cycle multiply in some circumstances. The instruction stream itself may be pipelined to reduce the effective instruction execution time. Finally, separate program and data memories allow simultaneous access to new instructions and data.

The paper is organized as follows: Section II describes the general architecture. Section III discusses the data routing in the chosen architecture, and Section IV gives a detailed description of the machine. Section V estimates its performance, and Section VI discusses some of the implementation issues that have not been addressed in this study. Finally, two Appendices describe architectural variations and a DMA port for the machine.

## II. ARCHITECTURAL OVERVIEW

A basic architecture which satisfies the above goals evolved through a combination of techniques. Some candidate machines were derived by posing an a priori instruction set, and others by postulating an architecture at the outset. In either case, both a generalized instruction set and related basic data routing concept were formulated. An instruction pipelining technique for each machine was developed and preliminary performance data for ECL 100K devices was used to estimate the throughput and data access capabilities of each machine. Finally, the apparent size, complexity, throughput, and programming ease of each candidate as well as intuitive feel were used to choose a final design.

Several basic approaches were examined by the above procedure. Array processors were found to be too difficult to program and performed non-array oriented processes very inefficiently. This suggested that a general-purpose structure would be more desirable. A three-address main-memory-to-main-memory structure, while very easy to program, was seen to require two copies of very fast data memory (about 800 total chips) which made the design too complex. Therefore a compromise architecture with a register file interposed between the main data memory and the arithmetic/logic unit (ALU) was chosen. A large register file implementation ($\geq$ 64) was projected to be too slow and seemed to require indexed addressing techniques to be used effectively. A small register file ($\leq$ 16) was judged to have inadequate capacity to effectively reduce the required main memory data flow. A multiple ALU strategy was tried in conjunction with the register file and found to afford only a moderate throughput improvement since a lengthened machine cycle was required to accommodate the additional data flow. Finally, a single ALU machine featuring a 32-word register file (Fig. 1) was developed as the best compromise in meeting the design goals.

## III. DATA ROUTING CONCEPTS

The conceptual architecture is shown in Fig. 1. (The topics of indexing, I/O and status will be considered later.) This machine can implement an instruction of the form

$$M_a \rightarrow R_b \qquad R_c \otimes R_d \rightarrow R_e \qquad R_f \rightarrow M_g \tag{1}$$

where $M_{a,g}$ represent any data memory locations, $R_{b,c,d,e,f}$ represent any register file locations, $\otimes$ denotes an arbitrary ALU operation, and $\rightarrow$ denotes the direction of the data transfer. Therefore, this instruction represents three operations: a memory to register transfer, an ALU operation, and a register to memory transfer. Said more simply, this corresponds to a memory read, an ALU operation, and a memory write. (The *conceptual architecture* is assumed to be capable of both reading and writing the data memory in a single instruction for illustrative purposes. This capability will be modified in the proposed real machine.) A possible instruction pipeline sequence for this machine is:

$$\begin{array}{|c|c|l|} \quad f \quad & \quad d \quad & \begin{array}{l} M_a \rightarrow R_b \\ R_c \times R_d \rightarrow R_e \\ R_f \rightarrow M_g \end{array} \end{array} \tag{2}$$

$$\begin{array}{c} \text{machine} \\ \mid\leftarrow \text{cycle} \rightarrow\mid \\ \rightarrow \text{time} \end{array}$$

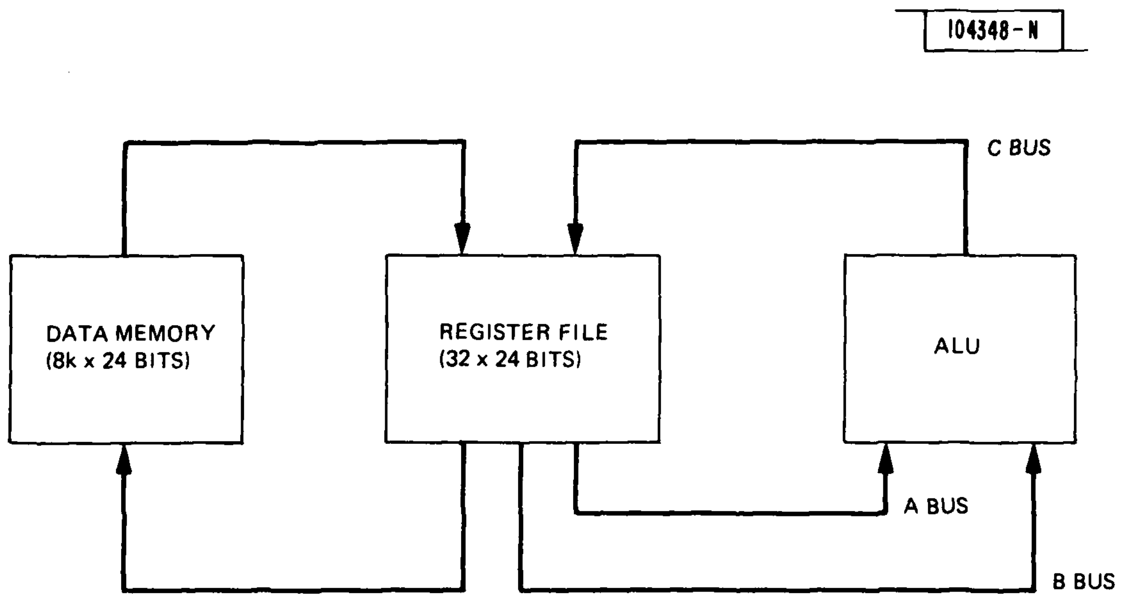


Fig. 1. Basic architecture.

where f denotes the instruction fetch, d the instruction decode and |
the pipeline epoch boundaries (all epochs are of the same duration).
Note that in a pipelined machine, a new instruction is started each cycle, and each instruction execution requires as many cycles as epochs in the pipeline. Thus, in this case, three instructions are in some stage of execution during any single cycle. All memory and register reads occur at the start of the final epoch, and all memory and register writes occur at the end of the final epoch. Therefore the result of each individual data operation is not available until the next instruction. For example, a program to add two numbers in memory and place the result back in memory might consist of the four instructions:

| | | | |
|---|---|---|---|
| $M_a \rightarrow R_a$ | -- | -- | (3a) |
| $M_b \rightarrow R_b$ | -- | -- | (3b) |
| -- | $R_a + R_b \rightarrow R_c$ | -- | (3c) |
| -- | -- | $R_c \rightarrow M_c$ | (3d) |

where the dash indicates no operation, and the subscripts indicate a particular location in memory or the register file. (The location $M_x$ is not equal to the location $R_x$ but is used here as a convenient way to follow an item of data as it is transferred between the memory and the register file.) In a real application the open instruction fields might be used as portions of preceding or following instruction sequences. Such interleaved programs are unnecessarily difficult to write and debug and become inefficient in the case of conditional branches.

A second pipeline to interpret the instruction format shown in (1) is:

$$\left|\quad f \quad\right|\quad d \quad\left|\; M_a \rightarrow R_b \;\right| R_c \otimes R_d \rightarrow R_e \left|\; R_f \rightarrow M_g \;\right| . \qquad (4)$$

This now permits data to flow within each instruction. The single add shown in (3) can now be programmed as follows:

$$M_a \rightarrow R_a \qquad -- \qquad -- \qquad (5a)$$

$$M_b \rightarrow R_b \qquad R_a + R_b \rightarrow R_c \qquad R_c \rightarrow M_c \qquad (5b)$$

Compared to the program of (3), this program is much easier to write, dovetails more easily with other operations, is more efficient near conditional jumps, and uses less program memory. Note that the memory, register file, and ALU perform the same operations in either case. The second case is more efficient since the pipeline naturally provides much of the desired sequencing of operations.

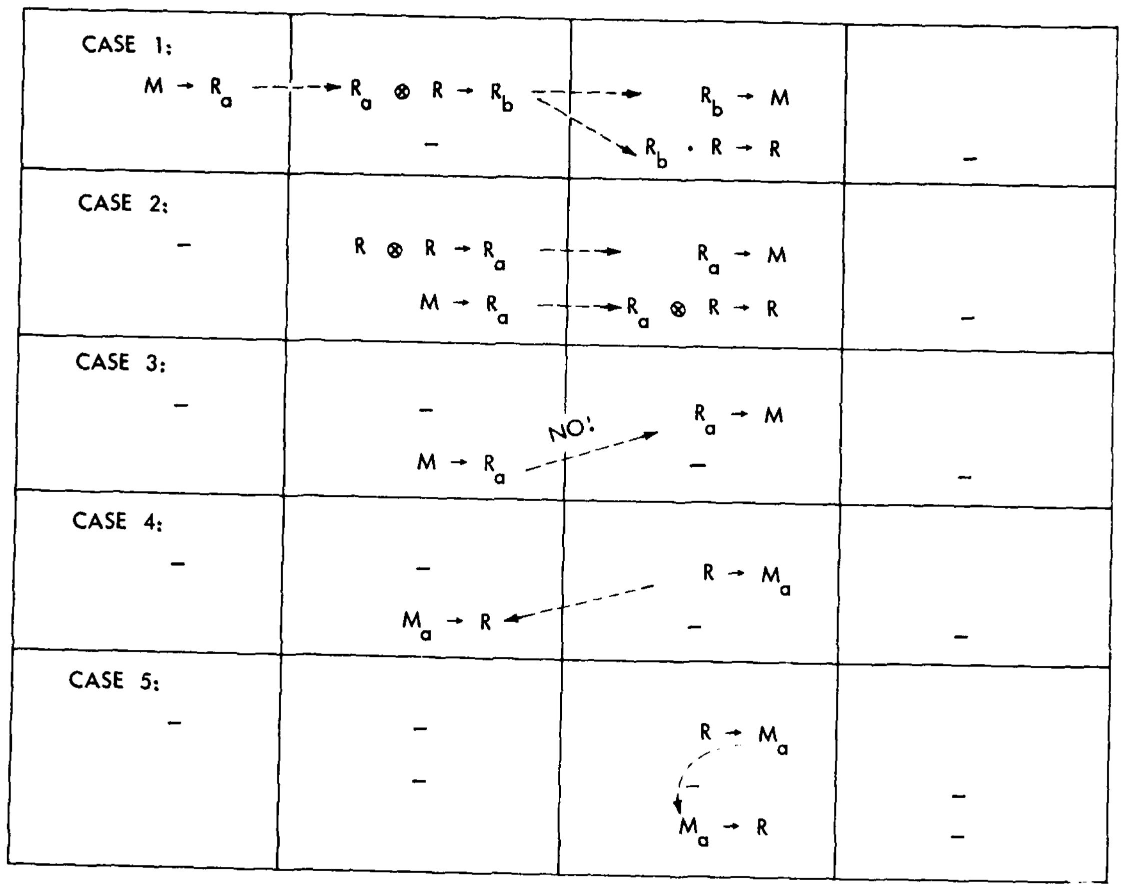
The advantages of the longer pipeline, however, are not free. The long pipeline introduces delays in the visibility of status data for conditional branches and delays execution of these operations. It could also introduce undesirable data routing patterns. For instance in the program

| | | | |
|---|---|---|---|
| -- | -- | $R_a \rightarrow M_y$ | (6a) |
| $M_x \rightarrow R_a$ | -- | -- | (6b) |

the read phase of (6b) occurs simultaneously with the operate phase of (6a) causing the datum in $M_x$ to be deposited in $M_y$ - an apparent reversal of the instruction sequence. To simplify programming, the pipeline must be hidden from the programmer. (Experience with the LDSP has shown that if the pipeline is visible in only a few simple, categorically defined ways, users will not only tolerate the pipeline, but will frequently use the otherwise wasted instruction cycles to advantage.)

The best way to hide the data-operation pipeline is to make the machine appear as if it were totally sequential; i.e., each instruction is executed in natural order - memory read, operate, and memory write - and each instruction is completed before the next is begun. Implementation of this programmers' model requires several special data routing conventions (Fig. 2). Clearly, data must flow from the memory read to the operate phase and from the operate phase to the memory write (Case 1). The results of an operate phase must also be available to the next operate phase (Case 1) unless superseded by a memory read (Case 2). The reversed

104349-N



NOTES: DOTTED ARROWS INDICATE DATA FLOW.
ONLY EPOCHS 3, 4 and 5 OF THE PIPELINE OF FIG.4 ARE SHOWN

Fig. 2. Data routing conventions for apparent sequential behavior. Note: Dotted arrows indicate the data flow. Only epochs 3, 4, and 5 of the pipeline of (4) are shown.

data flow of (6) must be blocked (Case 3) and a memory write must appear to be completed for any succeeding memory read (Cases 4 and 5). These special data routing conventions, as will be shown later, imply only a small amount of extra hardware. Control could be implemented by selected comparisons of register (Cases 1, 2, 3) and memory (Case 4) addresses. Case 5 is handled by the data memory write queue detailed in Section IV B.

If the data memory is incapable of supporting a full load of simultaneous reads and writes, this lengthened pipeline requires a write queue. A sequence of writes followed by a sequence of reads will require the data memory to simultaneously read and write for two cycles. A sequence of reads followed by a sequence of writes will require no memory operations for two cycles. Thus, a write queue can be loaded during cycles requiring simultaneous reads and writes and unloaded into the memory during cycles requiring no operation. The queue need only be two deep per write channel. If the queue is to be invisible to the programmer and handle data routing Case 5 (Fig. 2), an alternate (bypass) data path must be provided from the queue to the memory read bus.

## IV. DETAILED ARCHITECTURE

### A. General Description

The overall data architecture is shown in Fig. 3. Its general characteristics will be discussed in this section, and the details of each subsystem will be discussed in the sequels. The design presented here is one member of a family of three. The differences among variants are primarily related to the data memory. The three machines are:

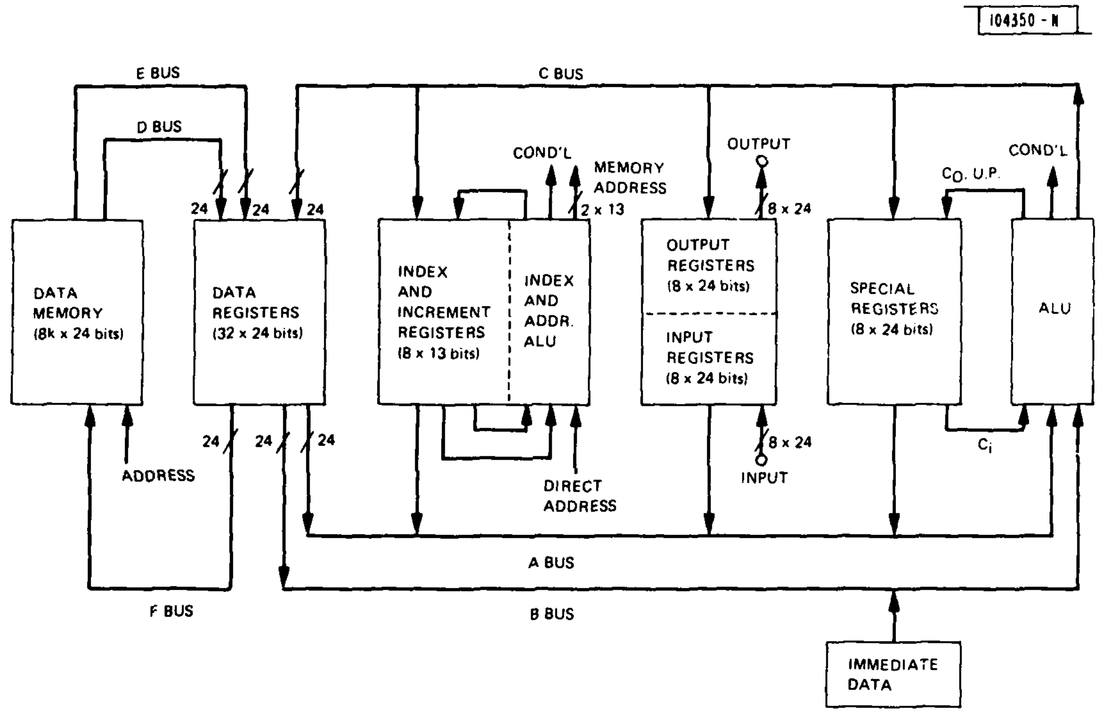

104350 - N
E BUS
C BUS
D BUS
COND'L
MEMORY ADDRESS
OUTPUT
COND'L
$C_O$ U.P.
24
24
24
2 x 13
8 x 24
DATA MEMORY (8k x 24 bits)
DATA REGISTERS (32 x 24 bits)
INDEX AND INCREMENT REGISTERS (8 x 13 bits)
INDEX AND ADDR. ALU
OUTPUT REGISTERS (8 x 24 bits)
INPUT REGISTERS (8 x 24 bits)
SPECIAL REGISTERS (8 x 24 bits)
ALU
24
24
24
8 x 24
ADDRESS
DIRECT ADDRESS
INPUT
$C_i$
A BUS
F BUS
B BUS
IMMEDIATE DATA

Fig. 3. Detailed architecture.

one-memory transfer (one memory read or write per instruction), 1.5-memory transfer (two reads or one write per instruction) and the two-memory transfer machine (two reads, two writes or one read and one write). The 1.5-memory transfer machine is presented since it is believed to represent the best compromise between hardware complexity and performance of the three. Details of the other alternatives are presented in Appendix A.

In describing the architecture, note that the data memory communicates only with the data register file, thereby isolating the data memory from the high data rates required by the ALU. The ALU is, however, capable of generating and consuming data faster than it can be deposited or accessed from the main memory. Benchmark analyses have shown that for typical speech signal processing algorithms a number of intermediate results are generated, and that storage of these intermediate results in the register file tends to relax memory data flow requirements sufficiently such that no data access overhead penalties are incurred in the inner loops of these algorithms. This allows the data memory to be implemented with dense but comparatively slow memory components.

The register file complex provides the intermediate result storage and main data routing control for the machine. The register file itself is implemented with small, very high-speed memory chips which can support up to seven register read or write cycles per machine cycle. The sequence of register operations within a single machine cycle, in conjunction with the extra data paths surrounding the memory proper, supports the data routing conventions of Fig. 2, Cases 1-4.

The indexing, I/O, status and special registers are situated between the A and C buses. These are accessed through the A bus (source) and C bus (destination) addressing mechanisms. Sixty-four addresses are associated with the A and C buses and 32 addresses with the B, D, E, and F buses (6 and 5 address bits, respectively). Thus these "extra" registers appear to the ALU as data registers and require a minimum of specialized instruction op codes for access and control. At the same time, the "extra" registers are not required to support the high data access rates of the register file.

The indexing and addressing complex provides the parallel memory addressing capability as well as a generalized index register auto-increment feature. Any of the eight index registers may be used to generate an indexed address for the data memory. Associated with each of the eight index registers is an increment register whose (signed) value is used to post-increment the index register contents if so instructed. One index register is equipped with a bit-reversed increment option to facilitate the bit-reverse data swap operation required in Radix 2 FFT implementations. All of the address generation and incrementation occurs in parallel with the rest of the instruction execution and costs no extra time. Full direct addressing is also permitted. If more complex address manipulations are required, the index and increment registers are immediately accessible to the ALU.

The proposed I/O system is of the duplex, fully buffered, vector interrupt type where the buffer registers are connected to the A and C buses. Therefore, the machine can transfer a datum between the I/O buffer and the data memory in a single instruction using a data register as an intermediate location, and, if desired, modify the datum with a single ALU operation. To simplify the control, no interrupt nesting is allowed. If all interrupt service routines are relatively short, this generally causes no problem in real-time interactive environments as has been demonstrated with the LDSP. A high-speed data memory port could also be incorporated as outlined in Appendix B.

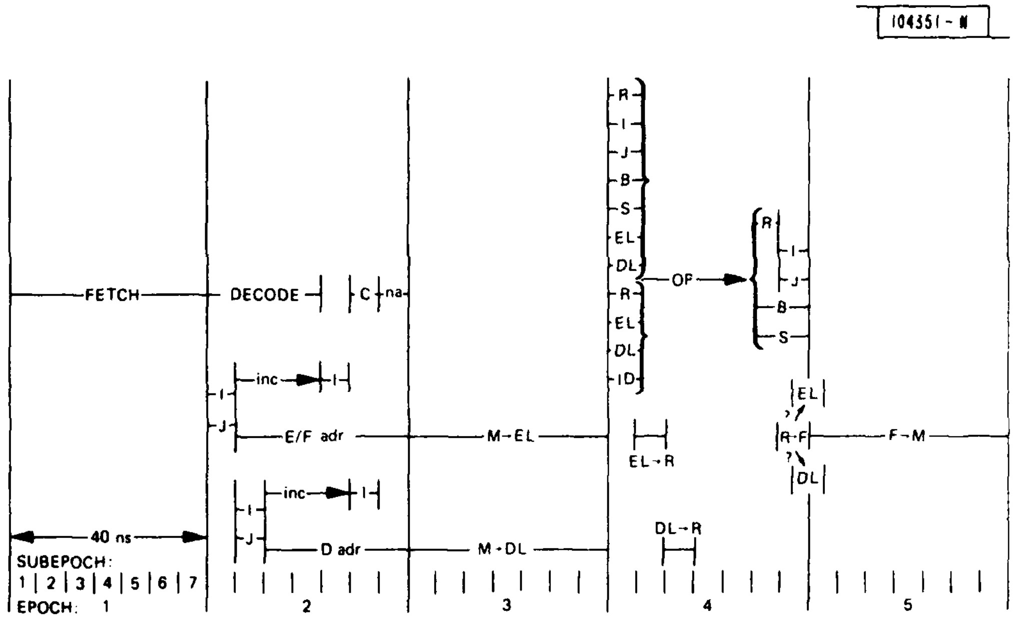
The special registers would include status information (e.g., the carry-in ($c_i$) bit, the ALU conditional and overflow flags), the multiplier upper (24-bit) product, the console data switch/light registers, the paths for loading the program memory, and control flags that need to be accessible to the program.

The ALU accommodates integer, floating point and logical operations. It is configured to accept two arguments simultaneously frcm the A and B buses and to route results to the C bus. Any arguments provided as immediate data from the program instruction word are entered on the B bus. The divide operation has been specifically excluded since fast divide hardware is complex and rarely essential in signal processing. Simple and fast programmed divide algorithms exist and can minimize the loss of throughput in the majority of cases.

The data word length, as mentioned earlier, is 24 bits. Experience with the LDSP, which has a 16-bit data word, indicates the need for more precision. Twenty-four bits was chosen as a compromise between accuracy, quantity of hardware, and speed. This choice also supports a reasonable floating point data format comprising a 16-bit signed fraction and an 8-bit signed exponent.

The detailed instruction pipeline diagram for the machine is shown in Fig. 4. The instruction fetch occurs in epoch 1. In epoch 2, the decode (micro-store ROM read) and indexing start immediately. Index register incrementation and indexed addressing start with the assumption that both modes are invoked. The actual decision as to which addressing option is called for develops in parallel and need be available only in time to prevent or proceed with the index register update. The (dual) data memory read occurs in epoch 3. The data routing indicated in Fig. 2, Cases 1-4 and the ALU operation takes place in epoch 4. Finally, the memory write operation occurs in epoch 5. For execution of a conditional branch, the ALU status (epoch 4) is sensed by the branch logic in epoch 2 of the second subsequent instruction (i.e., a 1-instruction latency). The new address is placed in the program counter at the end of epoch 2 in time for the fetch of the second subsequent instruction (i.e., another 1-instruction latency). Therefore, the programmer must wait one instruction before the ALU condition of interest becomes visible and one more instruction for the branch to take effect as in the following program:

```
ALU operation                                    (7a)
no-op                                            (7b)
conditional jump                                 (7c)
no-op                                            (7d)
(instruction at branch address if condition met). (7e)
```



adr = DATA MEMORY ADDRESS
B = I/O REGISTER
C = JUMP CONDITIONAL
DL = D BUS LATCH
EL = E BUS LATCH
F = F BUS
I = INDEX REGISTER
ID = IMMEDIATE DATA
inc = INDEX INCREMENT: I = I + J
J = (INDEX) INCREMENT
M = DATA MEMORY READ OR WRITE
na = NEW PROGRAM COUNTER ADDRESS (Jump)
OP = ALU OPERATION
R = DATA REGISTER
S = SPECIAL REGISTER
? = CONDITIONAL ON DATA ROUTING RULES
} = READ ONE OF
{ = WRITE ONE OF

Fig. 4. Pipeline timing diagram.

In most cases, the no-op instructions at (7b) and (7d) can be replaced with useful code.

While the control logic for the machine has not been worked out in detail, the general structure is clear. The instruction OP code is mapped into a set of control signals using a ROM micro-store. The control mechanism consists of a combination of the micro-store output and actual micro-coding of the instruction proper. The instruction register is itself pipelined to match the sequencing implied by Fig. 4. The instruction is composed of three fields: an ALU control field and two memory-control fields as shown in Fig. 5. Any unused register address bits in the ALU control field can be used to microcode suboptions on the basic instruction or to control the immediate data generator on the B bus. The corresponding assembly language forms are the following:

$$M_a \rightarrow R_b \qquad M_c \rightarrow R_d \qquad \left\{\begin{matrix} R_e \\ I_e \\ J_e \\ B_e \\ S_e \end{matrix}\right\} O_p \left\{\begin{matrix} R_f \\ \text{Immediate} \\ \text{Data} \end{matrix}\right\} \rightarrow \left\{\begin{matrix} R_g \\ I_g \\ J_g \\ B_g \\ S_g \end{matrix}\right\} \qquad R_h \rightarrow M_i \qquad (8a)$$

and

$$M_a \rightarrow R_b \qquad \text{branch (conditional) address} \qquad R_c \rightarrow M_d \qquad (8b)$$

where I = index, J = increment, B = I/O register, S = special register. Of the memory control fields, only two reads or one write may be called for in any single instruction.

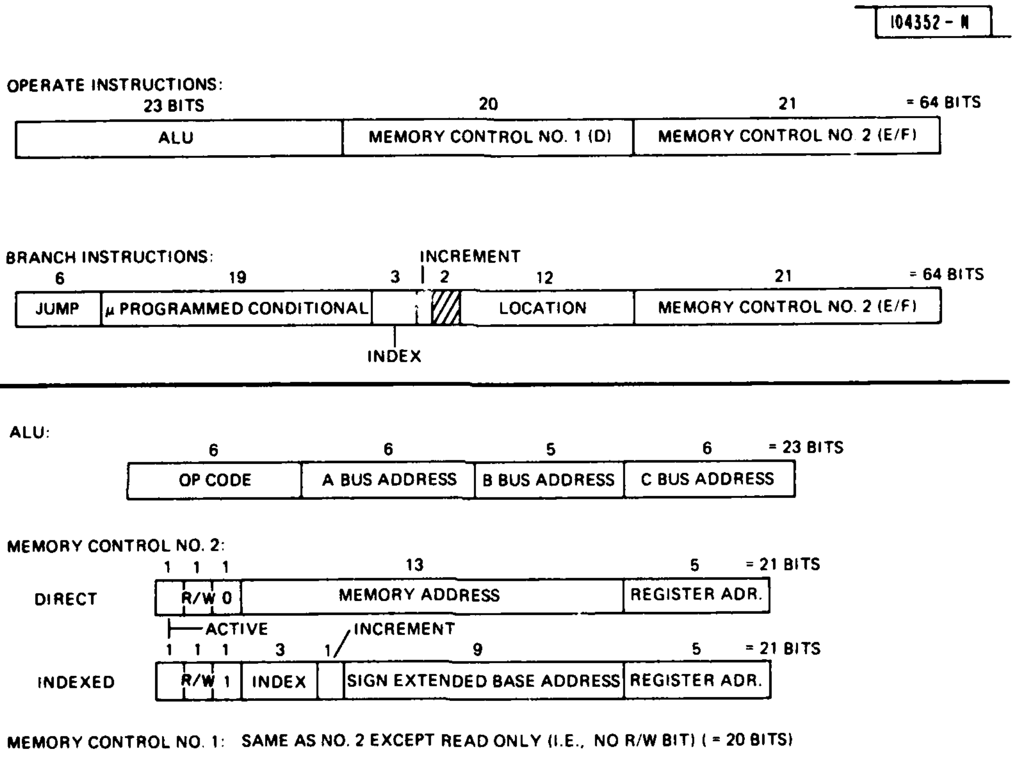


Fig. 5. Instruction formats.

The proposed instruction set is shown in Table I. Several instructions are unique. Since integer multiplies require two cycles, control is split into two successive instruction steps. The first starts the multiply, and the second offers the option of a combined add or subtract. The split instruction also offers two memory transfer opportunities. Several of the floating point arithmetic operations also require more time than is provided in a single cycle. These situations are handled simply by extending the epoch during which the operation takes place and are transparent to the programmer. The normalize instruction, instead of physically shifting the data, computes the number of shift places required. This count can be used as an argument to a subsequent shift instruction. The remaining instructions are a relatively typical set of integer, floating point, logical and control operations.

The major component subsystems are described in detail in the following sections.

## B. Data Memory

The data memory complex (Fig. 6) consists of the memory array proper, the write queue, and associated control logic. The memory array comprises two copies of an 8K x 24 bit memory capable of dual simultaneous read from or single write to the register file within a machine cycle. This is implemented with 96 F100470 4K x 1 bit RAMs[6] (35 ns maximum access time) and 91 F100112 Quad Drivers[5] for memory address fan-out. The write queue is required because the pipelining of the instructions may result in a request for both a data memory read and a data memory

TABLE I

INSTRUCTION SET

| Arithmetic | (integer) |
|---|---|
| ldi | load immediate |
| add | |
| sub | |
| subr | reverse subtract |
| smul* | start multiply |
| i | integer |
| d | fraction x 2 |
| f | fraction |
| h | fraction/2 |
| s | signed |
| u | unsigned |
| | end multiply* |
| mul | normal |
| mad | multiply add |
| msub | multiply subtract |
| msubr | multiply reverse subtract |
| norm* | normalize |
| shift* | shift |
| shifti | shift immediate |
| abs | absolute value |
| max | maximum |
| min | minimum |
| dadd | |
| dsub | |
| dsubr | double precision |
| dshift | |
| dshifti | |

| Miscellaneous | |
|---|---|
| in | I/O |
| out | I/O |
| loadMP | load program memory |
| nop | no operation |
| move | |

*see text

| Logical | |
|---|---|
| and | |
| or | |
| xor | exclusive or |
| cmp | compare |
| lshift | logical shift |
| lshifti | |
| set-bit | |
| clr-bit | |

| Floating Point | |
|---|---|
| fadd | (2 cycles) |
| fsub | (2 cycles) |
| fmul | (2 cycles) |
| fabs | |
| fmax | |
| fmin | |
| itof | integer to floating |
| ftoi | floating to integer |

Control
- jmp
  - conditional
  - switches
  - subroutine
  - subroutine return
  - interrupt return
  - kill (following instruction)
- load PC

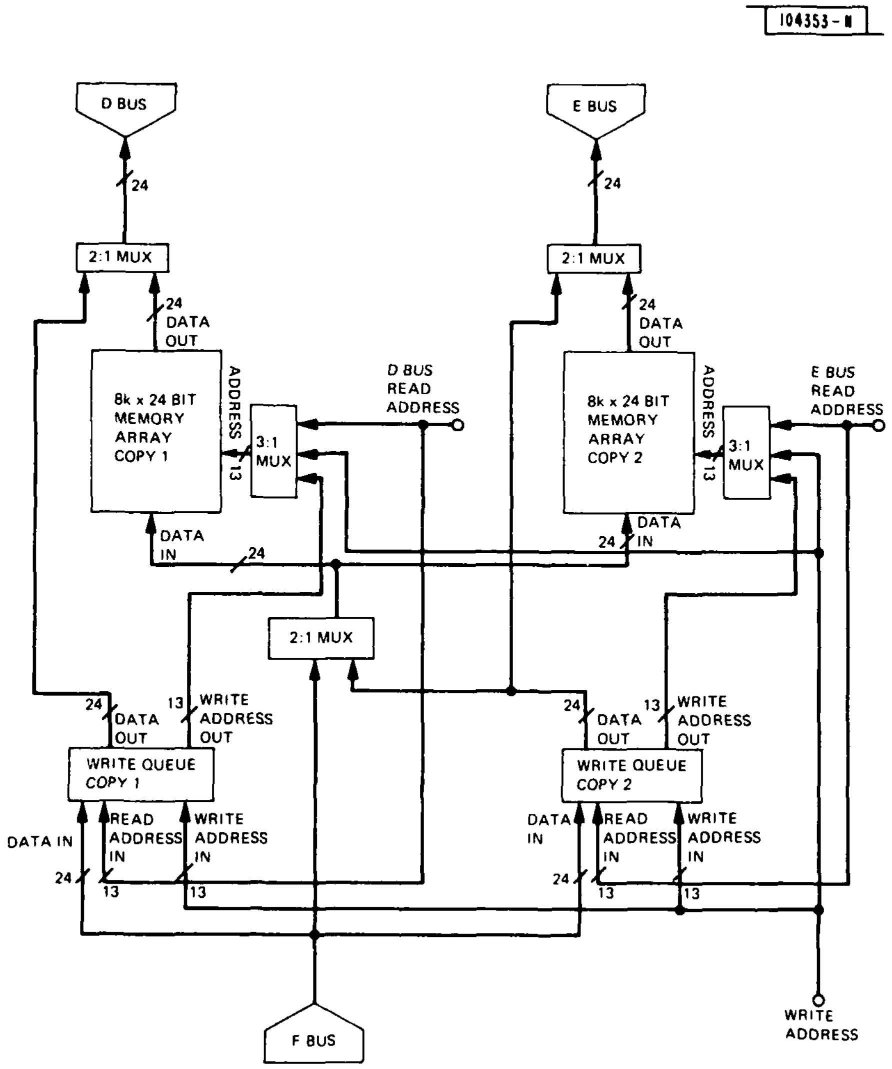


Fig. 6. Data memory system.

write within a single machine cycle despite the fact that the memory array can only read or write within a machine cycle. This is illustrated in Fig. 7 where the data memory write implied by instruction i occurs in the same machine cycle (c + 4) as the data memory read required by instruction i + 2. These conflicts are arbitrated by assigning the memory array to the read mode and postponing the write operation by storing the data and its data memory destination address in the write queue. This data is written into the actual memory array during a subsequent machine cycle when the memory array is not being used.

The write queue itself (Fig. 8) consists of a two-word data stack, a corresponding two-word memory address destination stack, and some address comparison logic which, as explained below, is required for arbitrating a second type of conflict arising when reference is made to data memory locations which are waiting to be updated by the write queue. The actual write queue registers and address comparison logic are implemented quite efficiently with the F100142 4 x 4 Content Addressable Memory[5] chip.

The detailed operation of the data memory complex within a machine cycle is described below for the four types of memory reference situations which can arise due to the pipelining of instructions: (1) read(s) and write, (2) read(s) only, (3) write only, (4) neither read nor write. Although two simultaneous reads may occur in a given machine cycle, for illustrative purposes only one read is described since the logical operations and concomitant hardware required for each are identical.

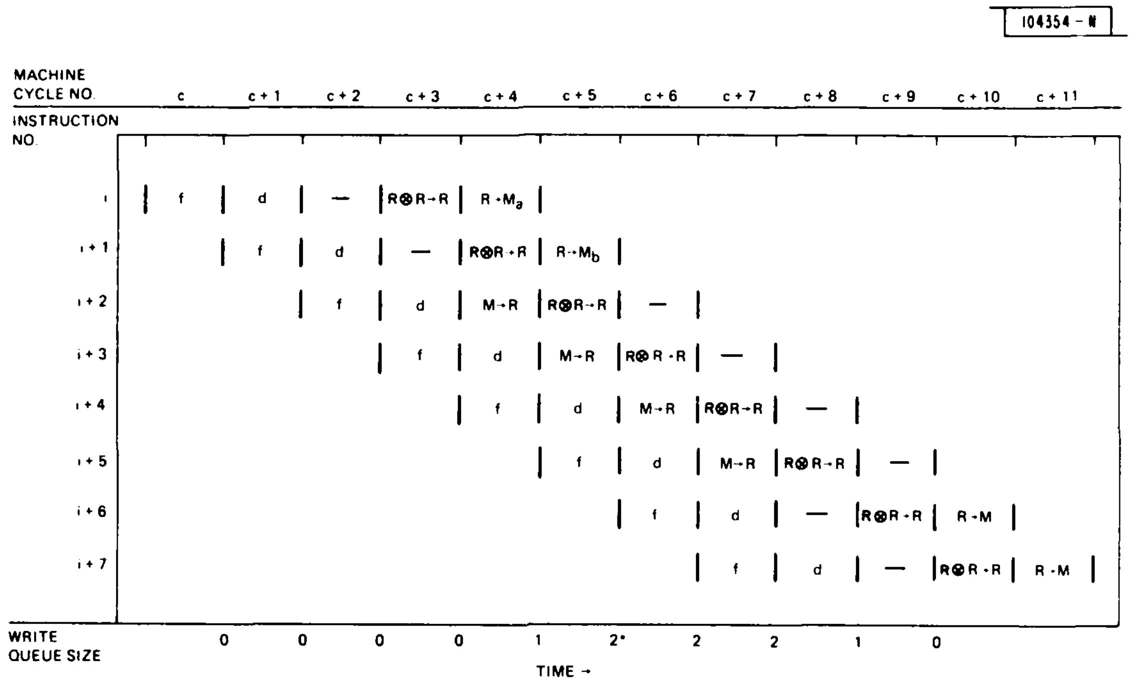


Fig. 7. Instruction sequence depicting write queue behavior.

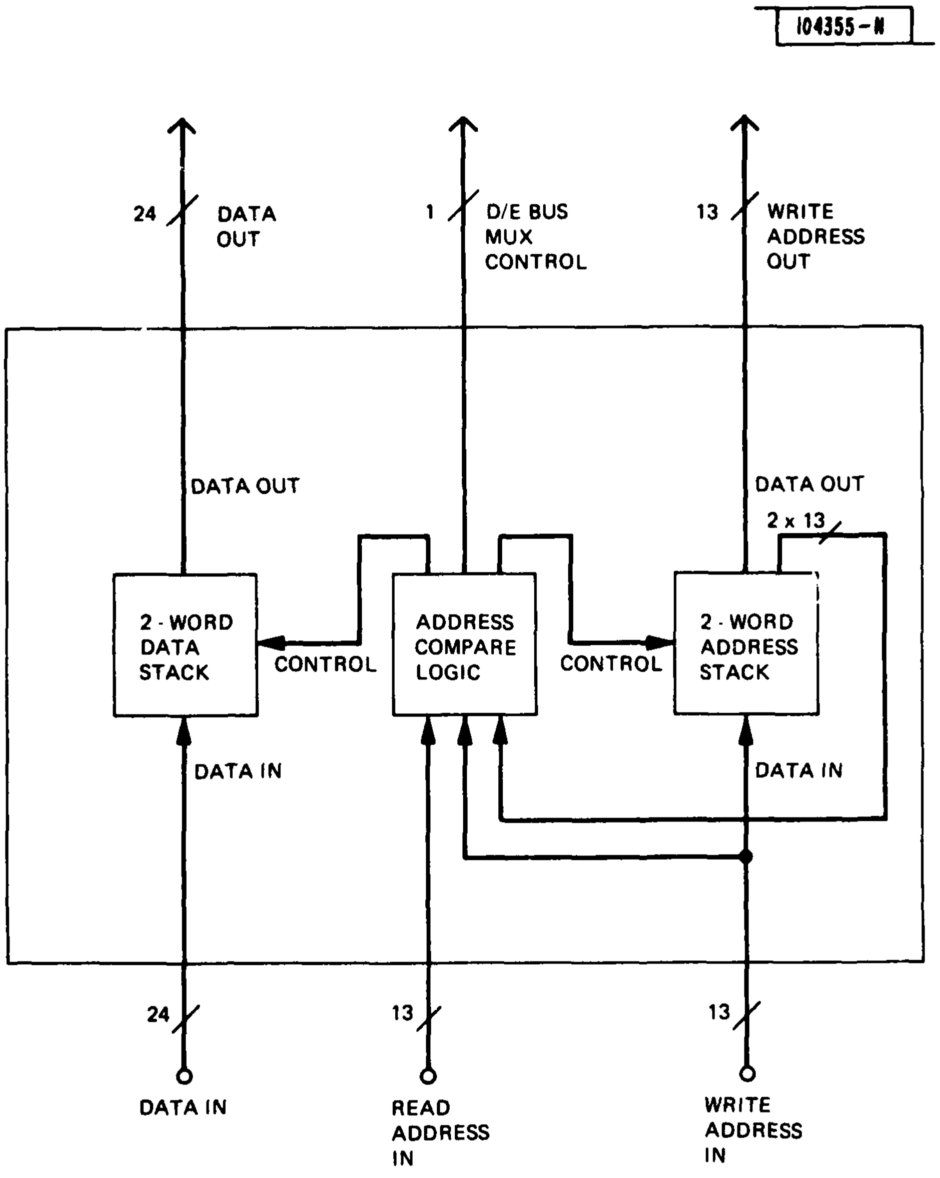


Fig. 8. Write queue (1 of 2).

For each case summarized below, the action taken depends firstly upon the size of the write queue data and address stacks (0, 1 or 2 words) and secondly upon the destination addresses in the address stack.

Case 1: Read(s) and write requested. The memory array is assigned to service the read request first, and the write data is placed in the write queue along with its data memory destination address. Nominally, the data is read from the memory array except when the queue contains updated data destined for the referenced location. This situation is detected by the queue address comparison logic, and the read data is obtained directly from the appropriate write queue data register thereby bypassing the memory. An additional comparison must be made to check whether the data about to be written into the write queue has the same destination as any other piece of data still on the queue. In this case the new data must overlay the old data on the write queue to obtain the correct update of the desired data memory location. Note that in such a machine cycle, the write queue may be accessed twice: once for the read (in the case of a read address match with a write queue address) and always once for the write. The write always obtains access to the queue in the first half of the machine cycle and the read in the second half. This is necessary for the case of a read from, and a write to, the same location in data memory so that the read operation will access the updated information from the queue (Fig. 2, Case 5).

Case 2: Read(s) only requested. This is a subset of Case 1. The memory array is used to service the read request, but again the write queue must be checked to see if it contains new data destined for the same address. If so, the read data is routed from the write queue data register instead of the memory array.

Case 3: Write only. The datum is nominally written into the memory array. An exception occurs when data is waiting in the write queue to be written into the same array address. In this case the new data is not written into the array but instead overwrites the old data in the write queue in order to prevent the old data from erroneously overwriting the new data on a subsequent cycle.

Case 4: Neither read nor write request. Since the array is nominally idle, the memory accepts any pending write from the queue.

An important feature of the write queue is that it need never exceed depth two. This is illustrated in Fig. 7 where a pipelined 8-instruction sequence is indicated which fills the write queue to its maximum and subsequently empties it. In machine cycle c + 4 the write/read conflict of instructions i and i + 2 results in the loading of the first datum onto the queue. Similarly, in cycle c + 5 the write/read conflict of instructions i + 1 and i + 3 results in the loading of the second datum onto the write queue. (It should be noted from Case 1 of above that if the memory reference addresses $M_a$ of cycle c + 4 and $M_b$ of c + 5 are the same, the second datum will overwrite the first datum on the write queue instead of being entered in the next available queue location.) At this point, any combination of instructions from i + 4 on will either maintain the queue size of two or empty it. In the case of

Fig. 7, instructions i + 4 to i + 7 result in a constant queue size of 2 through cycles c + 6 and c + 7 which subsequently decreases during cycles c + 8 and c + 9.

## C. Register File

As indicated earlier, special data paths within the data memory, register file and ALU subsystems must be supplied to support the data routing conventions of Fig. 2. The multiplexers and latches necessary to implement these paths are included as part of the register file complex and are described below along with the register file itself.

The register file comprises two copies of a 32 x 24 bit memory array consisting of 24 F100402 16 x 4 bit RAM[6] chips. These chips have a 5 ns maximum read access time which basically defines the machine's minimum subepoch.

Following the detailed instruction timing of Fig. 4 and the register file complex implementation of Fig. 9, it is seen that as many as two words are read from data memory onto the D and E buses and loaded into the D and E latches at the end of instruction epoch 3. This data is then written into the register file in the 2nd and 3rd subepochs of epoch 4.

Nominally, the inputs to the ALU are loaded from the register file onto the A and B buses during in the 1st subepoch of epoch 4. When the register(s) addressed as ALU inputs in epoch 4 are the same as those addressed for register file writes from data memory reads in epoch 3,

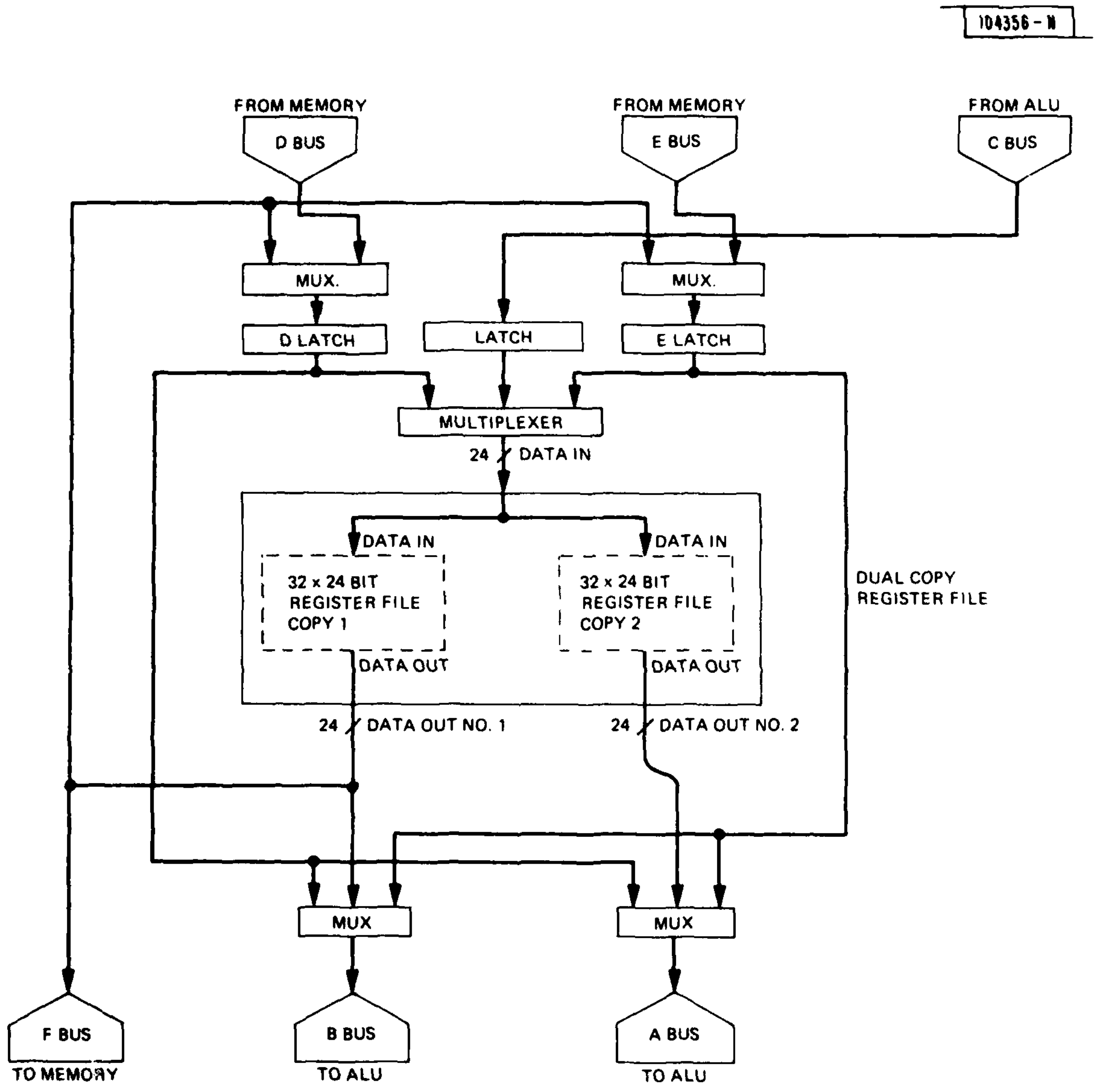


Fig. 9. Register file system.

the ALU inputs cannot be obtained from the register file at the time normally required since the required data has not yet been written into the register file. This situation is recognized through appropriate register address comparisons and rectified by bypassing the register file by multiplexing the D and E latches with the register file data outputs (Fig. 9) to load the A and B (ALU input) buses. The ALU operates during subepochs 2-5 in epoch 4, and the result is written back into the register file from the C (ALU output) bus in subepoch 6.

The above situation corresponds to case 2 illustrated in the data routing conventions of Fig. 2 and arises due to data access requirements _within_ a given instruction. When a read for an ALU operation references the same register as that addressed by the previous epoch's write from data memory, the data must bypass the register file because the register file has not yet been updated. The following situation corresponds to case 4 of Fig. 2 and arises due to data transfer requirements _between_ instructions: Assume that instruction i implies a write into data memory and is followed immediately by instruction i + 1 which requires a read from the same location in data memory. Since writes into data memory occur in epoch 5 and reads from data memory take place in epoch 3, the write operation of instruction i will have not have been completed at the time instruction i + 1 needs access to the data memory. This situation is recognized by data memory address comparison logic and is alleviated by routing the contents of the F bus (instead of the D or E bus contents) into the D or E latch. It can be seen that the F bus contains the desired datum since the register read onto the F bus (subepoch 7 of epoch 4) demanded by instruction i will have already occurred by the 1st subepoch of epoch 4 related to instruction i + 1. The data path is implemented by multiplexing the inputs to the D and E latches with the contents of the D and F buses and E and F buses respectively (Fig. 9).

## D. Indexing Unit

The responsiblities of the indexing complex (Fig. 10) include address computation for as many as two data memory references, index register post-increments, and direct modification of the index/increment register file through the ALU.

The data memory address computation employs parallel hardware to accommodate the most demanding case of a dual data memory read with indexed addressing. For maximum performance, both direct and indexed addressing are assumed at the start of epoch 2 (Fig. 4). The index registers are read as early as is possible in epoch 2 and added to the offsets coming from the memory control fields of the instruction. The deferred addressing mode decision from the instruction is implemented by multiplexing the computed address with the direct address from the memory control field of the instruction. Again, for maximum speed, the index registers are initially assumed to be incremented. The increment values are read from the index/increment file simultaneously with their corresponding index values used in the address computation detailed above and summed. The actual increment decision is again deferred until

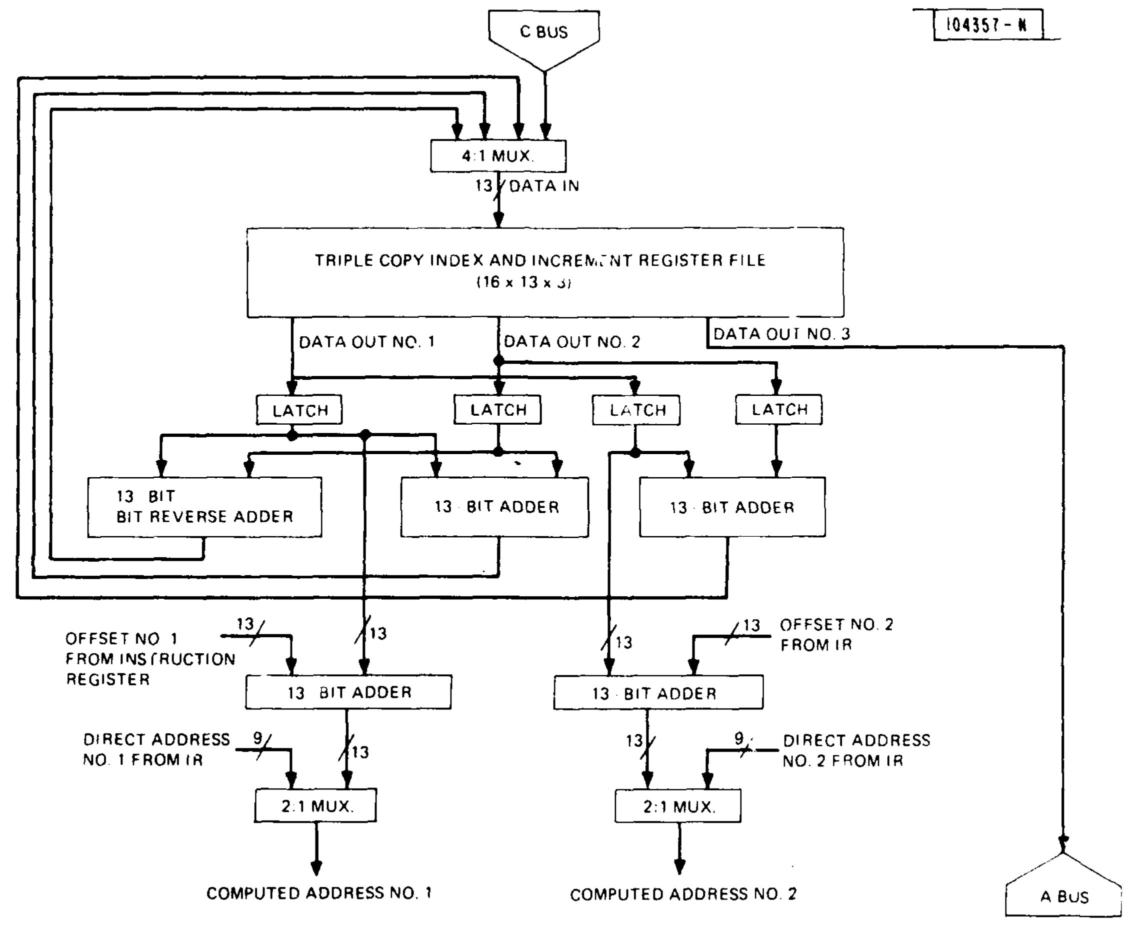


Fig. 10. Indexing unit.

the index file write subepoch. The indexing complex also contains a bit reverse adder, useful for Radix 2 FFT address generation, which replaces the conventional post-increment adder when activated. The bit reverse adder is invoked when the appropriate status register flag is set in conjunction with references to index register 0. As shown in Fig. 10, index/increment file outputs are conveyed to the ALU through the A bus. Inputs to the index/increment file from the ALU output (C bus) are sequenced to follow the two post-incremented index register writes. This provides a data transfer mechanism for instructions which use the ALU to modify the index/increment file.

The timing for the indexing complex (Fig. 4) is described below. The address computation and index register post-increment occur in instruction epoch 2. The first set of index and increment registers are read in the first subepoch, added in subepochs 2-4 and the resulting sum is written back into the index file in subepoch 5 to accomplish post-incrementation when elected. The index register contents are also added to the instruction register offset field and multiplexed with the direct address field, as described above, to develop the current instruction data memory address in subepochs 2-7. The same set of operations, delayed by one subepoch, occurs for computation of the second data memory address and index register post-increment in the case of a dual data memory read.

Since the index/increment register file has the same connectivity to the ALU as do the data registers (Fig. 3), it can be modified in epoch 4 in a corresponding way. The index or increment register to be modified is read on subepoch 1 of epoch 4 and written on subepoch 7. It should be noted that when instructions are pipelined, as many as 3 simultaneous reads from the index/increment file may be required in subepoch 1 of a machine cycle resulting in the need for the triple copy of the file.

## E. Arithmetic/Logical Unit

This section describes the hardware implementation of the Arithmetic/ Logical Unit (ALU) in two parts. Part 1 of the ALU (Fig. 11) accommodates the 24-bit integer add/subtract, multiply, multiply-add/subtract, maximum, minimum, and absolute value operations, and the floating point multiply operation. Part 2 of the ALU (Fig. 12) realizes the floating point add/subtract, maximum, minimum, and absolute value operations, the fixed-to-float and float-to-fixed point conversions, the (integer) normalization shift count derivation and programmed shift operations.

Parts 1 and 2 share 24-bit wide latches accepting inputs from the A and B buses and an 8-way, 24-bit wide multiplexer connecting to the C bus. A second A bus latch is required by the Part 1 subsystem to acquire the third operand for the two-cycle multiply-add/subtract instruction.

The major functional portions of Part 1 include the 24 x 24-bit array multipiier, 24-bit ALU, 8-bit exponent adder, and various multiplexers. The array multiplier is based on the F100182 9-bit Wallace Tree Adder[6] and F100183 2 x 8-bit Recode Multiplier[6] components and requires a total

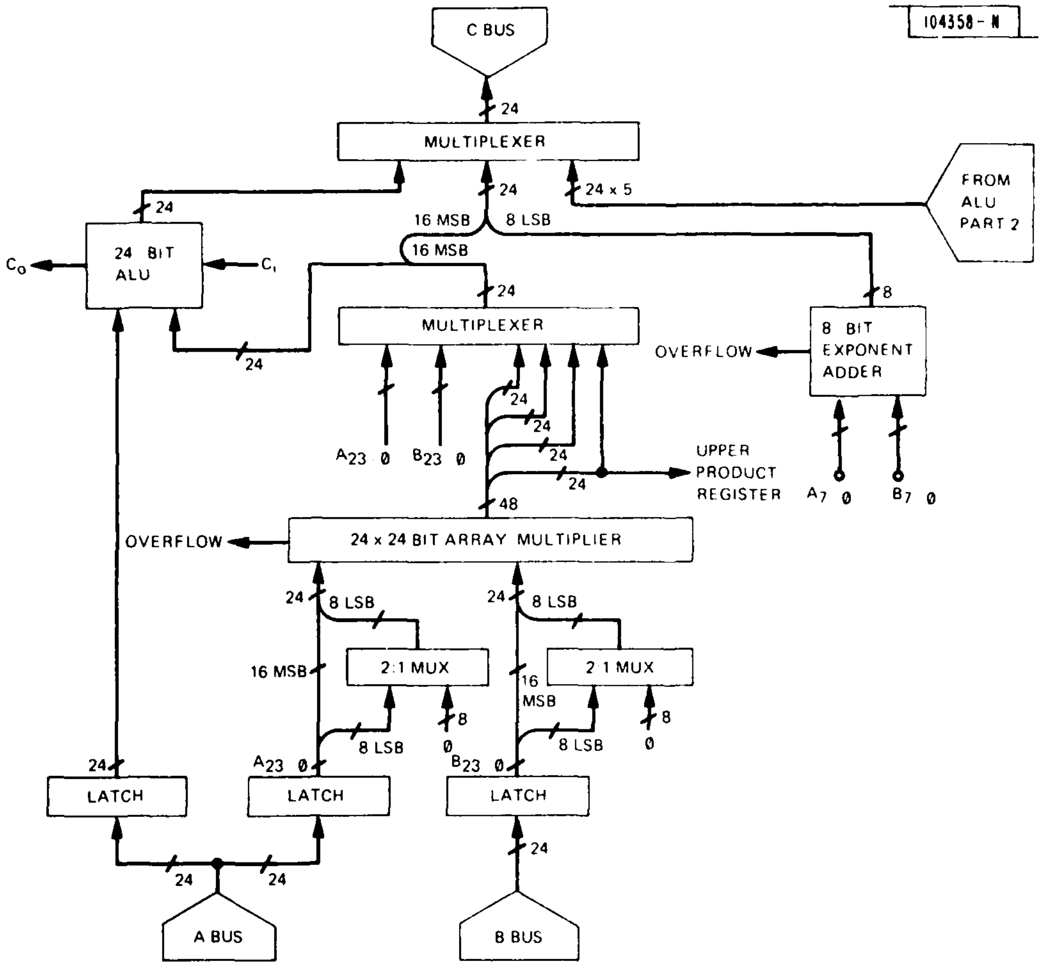


Fig. 11. Arithmetic/logical unit, Part 1.

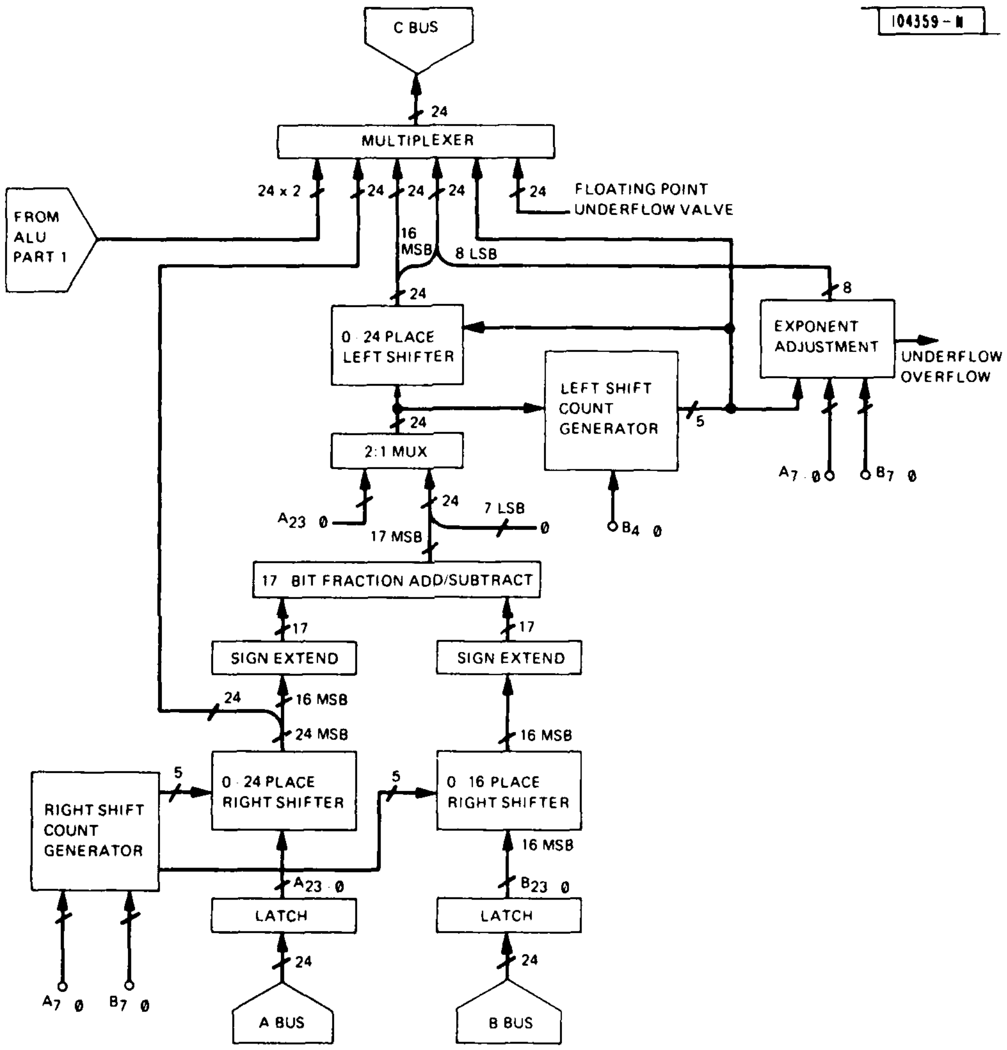


Fig. 12. Arithmetic/logical unit, Part 2.

of 127 F100K 24-pin packages.[5] The 8 least significant bits (LSBs) of the inputs to the multiplier are loaded with zeros to enable the array to function as a 16 x 16-bit multiplier for the fractional part of the floating point multiply. The 24-bit ALU is based on the F100181 4-bit binary/BCD ALU slice.[5] The multiplexer at the output of the multiplier plays several roles: (1) For the full 24 x 24-bit multiply it allows the programmer to interpret the input operands in fractional or integer formats and the appropriate 24 bit subset of the 48 output bits can be selected; (2) For the 16 x 16-bit (floating point) multiply it renormalizes the output when necessary (at most two left shifts); (3) Finally, it supplies the right-hand operand to the 24-bit ALU which can be the output of the multiplier in the case of the multiply-add/subtract instruction or the contents of the B bus latch for a simple add/subtract operation. The 8-bit exponent adder adds the exponents for the floating point multiply and re-adjusts the exponent for fraction normalization. Status outputs from Part 1 include overflow flags, carry-out and carry-in bits for the double precision integer add/subtracts and retention of the upper 24 multiplier bits in the upper product register for double precision integer multiplies. Part 1 is implemented with 181 24-pin F100K packages.

Part 2 of the ALU (Fig. 12) is essentially a floating point add/subtract unit with extra data paths to perform format conversions, normalization shift count derivation and programmed shifts. The floating point add/subtract proceeds as follows: The fractional parts of the inputs are aligned by right shifting the smaller input argument by the magnitude of the difference of the concomitant exponents. The resulting aligned fractions are then combined in the 17-bit fraction add/subtract unit and renormalized by left-shifting to obtain the fractional part of the result. Finally, the number of normalization left shifts is subtracted from one plus the larger input exponent to obtain the output exponent. Part 2 also recognizes instances of zero magnitude fractions and exponent underflow or overflow and routes a floating point underflow value through the C bus multiplexer.

The float-to-fixed point conversion operation uses the A operand right shifter. The fixed-to-floating point conversion process uses the left shift count generator, left shifter and exponent adjustment mechanism. The shift count generator develops the normalization shift count and produces a 5-bit quantity for later use in programmed shift operations.

The programmed left shift (indicated by a positive B bus shift value) is achieved by introducing the desired 5-bit shift count at the left shifter control lines from the B bus for A bus operands. Likewise, the programmed right shift (indicated by a negative B bus shift value) is obtained by providing the 5-bit right shift count at the right shifter control lines from the B bus for the A bus operand. The left and right shifters are implemented with the F100158 8-bit Shift Matrix chip.[5] Status outputs from Part 2 include underflow and overflow detection flags. Part 2 of the ALU is implemented with 74 24-pin F100K DIPs.

## F. Program Counter

The instruction execution sequence is controlled by the Program Counter (PC) complex (Fig. 13) by controlling the program memory read accesses for instruction fetches. The address information stored in the PC register is updated from the PC register input multiplexer each instruction fetch. In the absence of branches, interrupts, subroutine returns, and interrupt returns, the PC register is incremented by one. For branches the PC register is loaded from the branch address field of the instruction register. On interrupts the PC register receives the vectored interrupt address computed in the I/O complex. Returns from subroutine calls and interrupts are controlled with a 16-deep LIFO stack which contains subroutine and interrupt return location data. On subroutine branches and interrupts, the contents of the PC register plus one is pushed onto the stack. On returns from subroutine calls and interrupts the stack is popped into the PC register. A one-level status stack also stores appropriate status information during interrupts. To effect an indirect jump, the PC register may be loaded from the P register (cf., IV. I). Finally the PC register may be reset or loaded with the bootstrap initiation address.

## G. Program Memory

The program memory consists of 4K 64-bit words implemented with 64 F100470 4K x 1-bit RAM[6] chips (35 ns maximum access time) and 60 F100112 Quad Driver[5] chips for memory address fan-out. The read and write address lines come from the Program Counter complex (cf., Section IV. F). The

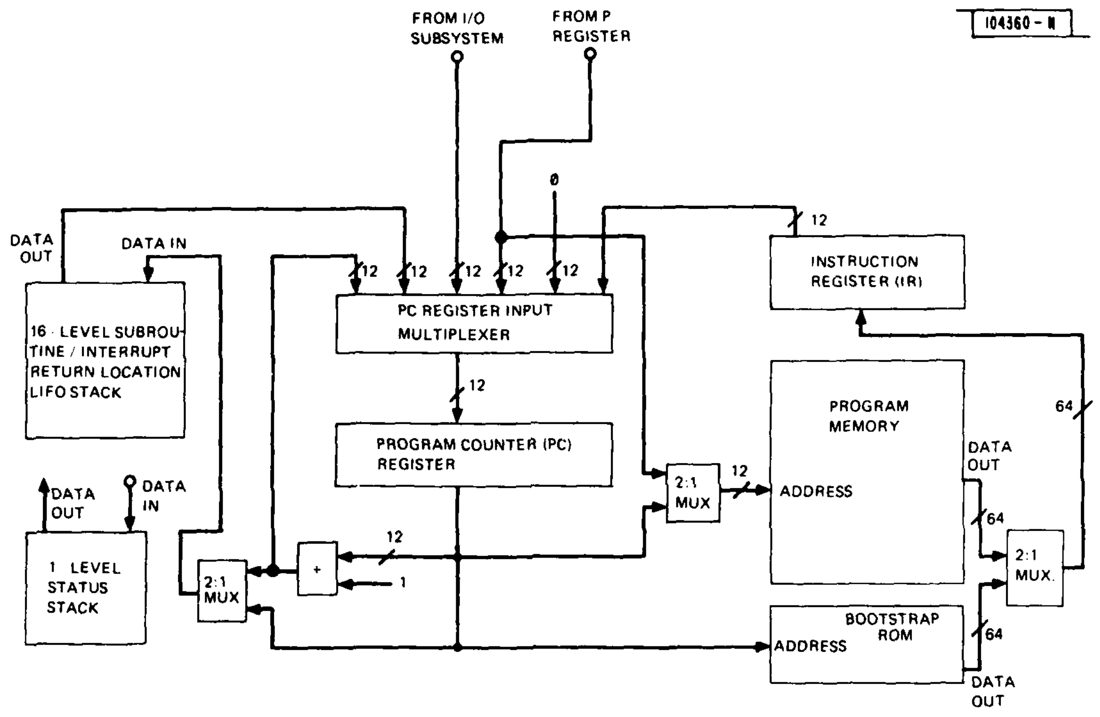


Fig. 13. Program counter system.

data output lines connect to the instruction register (cf., Section IV. H).

## H. Control

The machine control signals are derived from both the program memory instruction word directly as well as from a micro-instruction word read in the first four subepochs of epoch 2. The micro-instruction is stored in a PROM composed of F100416 (20 ns maximum access time) chips.[5,6] Due to the pipelined nature of the instruction execution process, control signals must also be delayed and buffered for as many as four machine cycles. The control structure is depicted conceptually in Fig. 14 by a series of registers storing the critical fields of the instruction and micro-instructions.

## I. Special Registers

The remaining machine registers, called the Special Registers, are as follows:

- status register (9 bits)
- upper product register (24 bits)
- P register (12 bits)
- 3 program memory load registers (total of 64 bits)
- console input/output register (each 24 bits).

The status register components are as follows:

- carry save bit (loaded from the carry-out and used as the carry-in of the integer ALU for double precision add/subtract)
- integer add/subtract overflow flag

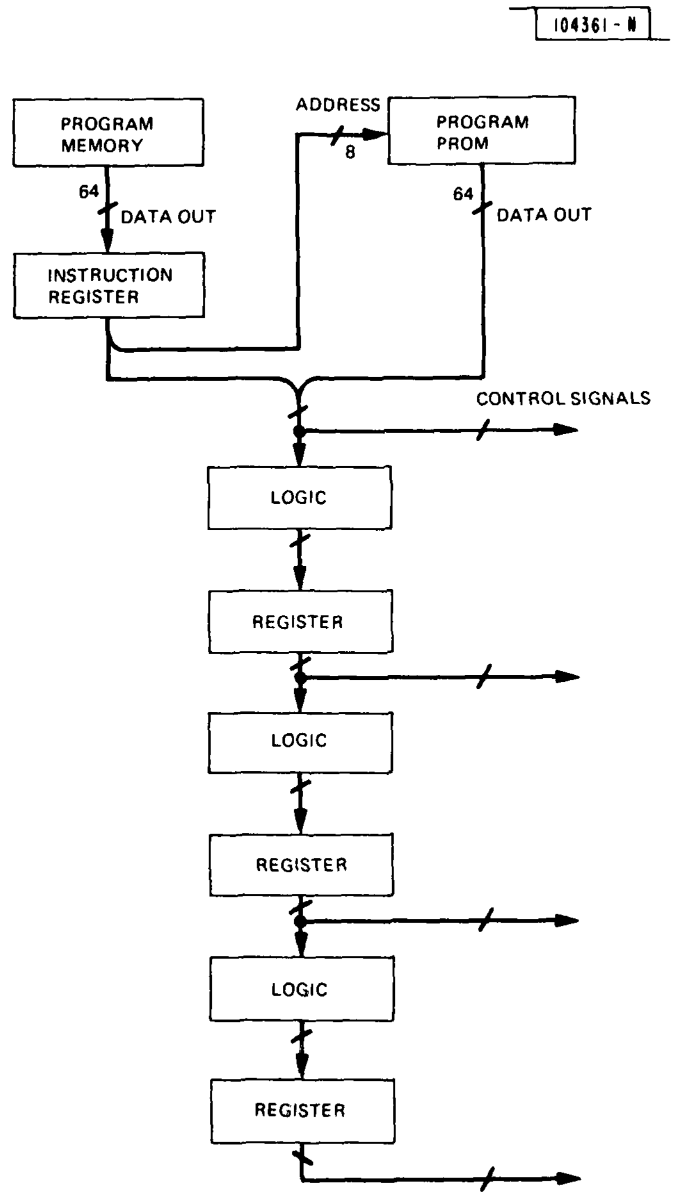


Fig. 14. Pipelined control structure.

- integer multiply overflow flag
- floating point overflow flag
- floating point underflow flag
- bit-reverse mode (programmed: designates bit-reverse increment mode for index register 0)
- subroutine/interrupt return location stack overflow flag
- master interrupt lockout (programmed)
- interrupt service lockout (set/reset by machine upon entry to or exit from interrupts).

The upper product register contains the 24 MSBs of the previous integer multiply and is used for double precision integer multiplies. Both the status and upper product registers are pushed on to a one-level status stack upon interrupt and popped upon return (see Section IV, F). The three program memory load registers are used as the data input buffers for initial program memory loading during bootstrap. The P register provides the program memory address for loading and indirect branches. Finally, the console register buffers input from console switches and output to console lights.

## J. Parts Count and Cost Summary

From the preliminary designs of the processor's subsystems, it is estimated that a total of about 1165 chips are required and could be accommodated with 7 wire-wrap boards (cf., VI) (Table 2). The total parts cost is estimated to be about $38,000 (Table 3).

TABLE II

DEVICE AND BOARD COUNT

| | MAJOR SUBASSEMBLIES | DEVICE SUBTOTAL |
|---|---|---|
| PROGRAM MEMORY | 4K x 64 BIT ARRAY: 64 4K x 1 RAMs<br>ADDRESS DISTRIBUTION: 60 DRIVER CHIPS | 124 |
| DATA MEMORY | 2 x 8K x 24 BIT ARRAY: 108 4K x 1 RAMs<br>ADDRESS DISTRIBUTION: 91 DRIVER CHIPS<br>WRITE QUEUE AND RANDOM LOGIC: 35 CHIPS | 234 |
| DATA REGISTERS | 2 x 32 x 24 BIT ARRAY: 36 16 x 4 RAMs<br>ADDRESS DISTRIBUTION: 10 DRIVER CHIPS<br>MUX'ING AND LATCHES: 52 chips | 98 |
| INPUT/OUTPUT | | 225 |
| ARITHMETIC/ LOGICAL UNIT | 24 x 24 BIT ARRAY MULTIPLIER: 127 CHIPS | 255 |
| INDEXING UNIT | | 63 |
| STATUS REGISTERS | | 48 |
| PROGRAM COUNTER | | 22 |
| CONSOLE | | 32 |
| μPROGRAM ROM, TIMING, CONTROL | | 64 |
| | TOTAL DEVICE COUNT | 1165 |

TOTAL BOARD COUNT 7

TABLE III

ESTIMATED PARTS COST

| | | |
|---|---|---|
| SEVEN WIREWRAP BOARDS @ $1,500 | | $10,500 |
| 1165 INTEGRATED CIRCUITS @ $15 | ≈ | 17,500 |
| MISC: POWER SUPPLY, ETC. | | 10,000 |
| TOTAL | | $38,000 |

## V. PERFORMANCE BENCHMARKS

Since a substantial portion of the processing time in many digital signal processing algorithms is devoted to the execution of orderly inner loops, a useful measure of a given processor's throughput can be inferred from the execution times corresponding to these inner loops. Furthermore, since these benchmark loops are generally implemented in a relatively small number of program steps, this evaluation does not require an inordinate amount of effort. These benchmarks not only provide an overall figure of merit for a particular processor design but play an interactive role in the design process itself.

Four examples important in digital speech and signal processing were chosen as shown below (Fig. 15a-d):

(a) the multiply-accumulate of a dot product

(b) the LPC lattice filter cell (2 multiplier form)

(c) recursive filter 2nd order section (2 poles and 2 zeros)

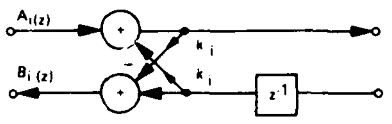
(d) the radix 2 FFT butterfly.

These four benchmarks were first coded for the LDSP as a reference. It was found that as few as 1/3 of the cycles of the resulting LDSP code could be ascribed to ALU operations. The remaining 2/3 were devoted to index register manipulation, loop counter maintenance, and data transfers. Therefore, for equivalent ALU speeds, an improved data architecture could potentially increase processor throughput by a factor of 3 through increased ALU utilization. With respect to the proposed machine, throughput ranging from 3.4 to 5.2 times that of the LDSP

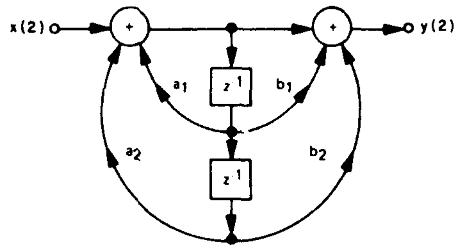
104362-N

$(\text{SUBTOTAL}) \leftarrow (\text{SUBTOTAL}) + x_n \cdot y_n$

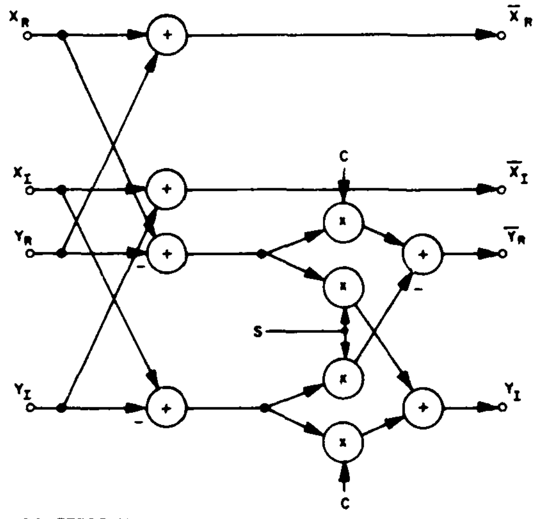
(A) MULTIPLY - ACCUMULATE OF DOT PRODUCT



(B) LPC LATTICE FILTER CELL



(C) RECURSIVE FILTER 2nd ORDER SECTION



(D) RADIX 2 BUTTERFLY

Fig. 15. Benchmarks.

(Table 4) is achieved for the four benchmarks. In part, this is accomplished by increasing ALU utilization to 100% with a suitably flexible data architecture as well as through the elimination of indexing operations and data access delays within the inner loops. The remaining throughput increase is achieved through the faster and more powerful Arithmetic/ Logical Unit structure. The fixed point add/subtract and standard logical operations are reduced from 50 ns in the LDSP to 40 ns while the fixed point multiply-add sequence is reduced from 150 ns to 80 ns.

The proposed machine was also benchmarked against existing array processors in execution of a 1024-point complex FFT (Table 5). Throughput greater than the proposed processor was achieved only by the Westinghouse PSP-X+ and Hughes PSP-E processor at the expense of reduced accuracy, long programming pipelines and specialized architectures. The proposed design's throughput was found to be greater than the FPS AP120B array processor at the cost of reduced accuracy but with the benefits of programming simplicity.

Finally, while the benchmark algorithms delineated here characterize a considerable portion of the digital speech processing computation time, much speech processing is characterized by non-structured non-iterative algorithms. The performance of array processors or architectures with highly pipelined arithmetic units degrades rapidly in such cases, but the design presented here maintains throughput as well as programmability.

TABLE IV

THROUGHPUT COMPARISONS FOR SELECTED BENCHMARKS (INTEGER ARITHMETIC)

| | LDSP EXECUTION TIME (NS) | PROPOSED MACHINE EXECUTION TIME (NS) | PROPOSED MACHINE THROUGHPUT (NORMALIZED TO LDSP) |
|---|---|---|---|
| DOT PRODUCT | 350 | 80 | 4.4 |
| LATTICE FILTER CELL | 550 | 160 | 3.4 |
| 2nd ORDER SECTION | 1650 | 320 | 5.2 |
| FFT BUTTERFLY | 1900 | 480 | 4.0 |

TABLE V

ARRAY PROCESSOR COMPARISON FOR 1024 POINT COMPLEX FFT

| PROCESSOR | DATA FORMAT | TIME (ms) |
|---|---|---|
| CSPI MAP-300 | 32 BIT FLOAT | 4.5 |
| FPS AP120B | 38 BIT FLOAT | 4.75 |
| ANALOGIC AP400 | 24 BIT BLOCK FLOAT | 7.4 |
| CD & A MSP | 24 BIT BLOCK FLOAT | 13. |
| SPS 81 | 16 BIT FIXED | 4.0 |
| WESTINGHOUSE PSP-X+ | 16 BIT FIXED | 0.51 |
| HUGHES PSP-E | 12 BIT FIXED | 0.26 |
| LDSP* | 16 BIT BLOCK FLOAT | 10. |
| PROPOSED PROCESSOR* | 24 BIT FIXED | 2. |
|  | 24 BIT FLOAT | 3.3 |

*GENERAL PURPOSE ARCHITECTURE

## VI. HARDWARE IMPLEMENTATION ISSUES

The basic elements intended for use in implementing the high-speed processor are those available in the Fairchild F100K[5,6] logic series. This is a line of ECL elements featuring propagation delays in the subnanosecond region (typically 750 picoseconds) and edge rates of approximately 1 volt per nanosecond. These parameters dramatically influence the packaging philosophy of the high-speed processor relative to that utilized in the LDSP for the F10K ECL logic line. In particular, there is a serious limitation on the length of open wire interconnections permissible with F100K logic elements. Thus, most connections are constrained to be effected via balanced twisted pair. Furthermore, to accommodate the F100K logic, a special wirewrap panel design[7] is an absolute necessity in terms of strict impedance control and maximum utilization of voltage and ground plane metalization area. Decoupling issues are also of paramount importance. An important fact relating to these special requirements is the existence of a very studied special high-speed wirewrap panel design. It is also known that a specialized set of support resources exist including a transportable, sophisticated CAD software package for board and system designs using F100K elements.[8] This CAD capability includes wirelist generation and debugging, placement and routing features, functional design libraries, hierarchical linking of functional design groups, and timing verification and emulation. An extensive interactive graphics package is also included as well as full documentation support. It is obvious that this level of support represents an immense potential savings in the engineering effort required to implement a large system employing F100K elements. The existence of such a powerful resource is expected to largely mitigate added demands imposed upon the system design problem by the ultra-high performance technology.

A factor having overriding impact on packaging technique is that of logic speed. Since the F100K parts feature extremely fast edge rates, precise impedance matching of every wired connection is a necessity in order to prevent noise and discontinuity effects from significantly reducing the system timing margins. The F10K series of integrated circuits was designed to be compatible with the basic wirewrap board environment by controlling edge speeds and impedance levels. The faster edge speeds of the F100K series imply modifications to the basic wire-wrap board design philosophy.

Since propagation delays in the F100K series are shorter by a factor of three than those of the F10K series, it might be expected that the speed of the new processor could be improved over that of the LDSP by a similar factor. However, a number of subtle system issues might prevent this potential from being realized in actual practice:

1. Device delays constitute a relatively small fraction of total system delay relative to signal propagation along twisted wirewrap pairs which comprise a vast majority of F100K interconnection paths.

2. The increased component count of the new processor over that of

the LDSP implies a physically larger piece of hardware.

3. The effect of possible increased fanout requirements due to the F100K technology must be considered. Since this is a critical factor, the assumptions for average fanout in the new processor have been predicated upon very conservative figures with respect to those used for the LDSP. Average fanouts for the new processor are reduced by a factor of approximately two. Should this ultraconservative approach prove unnecessary as the design evolves and matures, it could impact very favorably upon the projected system performance.

4. The new machine will require at least a factor of three more board area than the LDSP which implies increased average signal propagation delays. Furthermore, propagation delays over interboard cables are greater than those normally attributed to single-ended wire over groundplane. Therefore system partitioning is a critical issue.

Since the factors mentioned above cannot be quantified in detail until a final design for the system is developed and bench tests performed, they can be employed only to arrive at an educated estimate of system performance. This has been done in the case of system benchmarking as discussed earlier. Thus the comparative figures developed in the benchmarking exercise reflect consideration of the factors delineated above and represent a "best estimate" in the absence of detailed performance data. It is expected that the timing verification features of the CAD system will provide a more reasonable estimate of the system performance.

Another important factor is the tendency of memory element performance to double every year. Since memory element performance is so critical in the proposed architecture as per the register file, such an improvement would bear directly on overall system throughput.

Since a copy of the advanced wirewrap panel has been released to Lincoln Laboratory for internal use, it is expected that experimental prototyping will provide some answers to signal propagation behavior prior to the time that final designs are formulated as was the case for the LDSP. The experimental prototype will be used as a testbed for studying special signal distribution cases.

A number of different chip carriers are available in the F100K family. Initially, the 24-pin DIP packages were thought to be the best choice for use with the wirewrap board; however, recent results by other investigators[8] suggest the possibility of improved performance using flat packs in connection with special adapter sockets. The use of special sockets for the standard 24-pin DIP board design is a necessity in any case since some of the devices in the F100K series are in 16-pin packages. This is of particular significance in the case of the F100402[5] register file chips whose speed parameters define the basic subepoch of the processor. Studies have shown that approximately 200 picoseconds of additional delay is encountered in passing from chip carriers to the DIP sockets on the wirewrap panel. Since this is the same amount of time gained by use of the flat pack in preference to the DIP package, it is expected that use of 16-pin flat packs with chip adapters will result in approximately the same speed performance as specified for 16-pin DIP packages plugged directly into board sockets.

Fairchild has also introduced a special series of LSI chips in the F100200 family.[6] These components represent a very high level of chip integration but are packaged on a special 68-pin carrier which would probably be unsuitable for use on a wirewrap panel. The F100200 series presently consists of only four chips, three of which do not appear to be particularly useful in the proposed high-speed processor architecture. However, the F100223 programmable interface unit (PIU)[6] could be applied in the capacity of an I/O handshaking and buffering element. This would probably be housed on a printed circuit board attached via some special arrangement to the wirewrap board. It is expected that more potentially useful parts will be forthcoming in the F100200 family. This might suggest a complete printed circuit card design which could be integrated with a wirewrap card complex since the F100K and F100200 series are compatible.

The most significant departure for F100K design over lower technologies is in the area of clock and signal skews. Since the device parameters involve very high speeds and short propagation times, the wire delays constitute a significant portion of the communication time between elements in the system. Thus special care must be taken to see that clocks are distributed to the various parts of the system with closely matched skews. The partitioning of system elements is a matter of the utmost concern at both the inter-board and intra-board levels. The CAD system addresses all such issues, and it is hoped that it will minimize the time needed to complete this part of the processor design.

One area of somewhat relaxed concern is that of system thermal management. It is expected that an air flow similar to that employed in the LDSP (500 LFPM) should be adequate for the F100K design. In this way the ease of maintenance and accessibility achieved in the LDSP package should be preserved in the new design.

## VII. CONCLUSIONS

This design is subject to a number of unknowns, all of which relate to the use of the developing ECL 100K technology. For example, fanout versus time delay tradeoffs are not well characterized. The wiring is subject to transmission line effects. Several different package styles are available. Wirewrap boards suited to the logic family are rather restrictive (~200 sockets) resulting in potential layout difficulties. On the other hand, denser and/or faster memories will probably soon be available. Highly integrated I/O controller chips (e.g., the F100223) are available and may be usable in the I/O system. Thus much remains to be learned or solved before a detailed design of the machine can be completed.

This design exercise does indicate that it is feasible to conceptualize a machine which meets the design goals - high throughput with high programmability and small size. The goals are achieved by using a faster logic family and a carefully chosen architecture which is intended to strike a balance between parallelism and programming complexity. Additionally, the architecture proposed here is a general-purpose architecture which may be used with equal facility in both array- and non-array-oriented applications. Existing commercially available array processors, achieve similar throughput levels using high degrees of parallelism but at the cost of difficult programming and poor performance for non-structured computation.

This architecture may have applications beyond those proposed here. Given that its complexity is on the order of 100,000 gates not including the large memories, it will soon be possible to place the system on one or just a few VLSI chips. The cycle time of the machine would probably be slowed by a factor of at least five or ten in the case of MOS technology, but this would still represent a powerful machine for many applications.

# APPENDIX A

## DESIGN VARIANTS

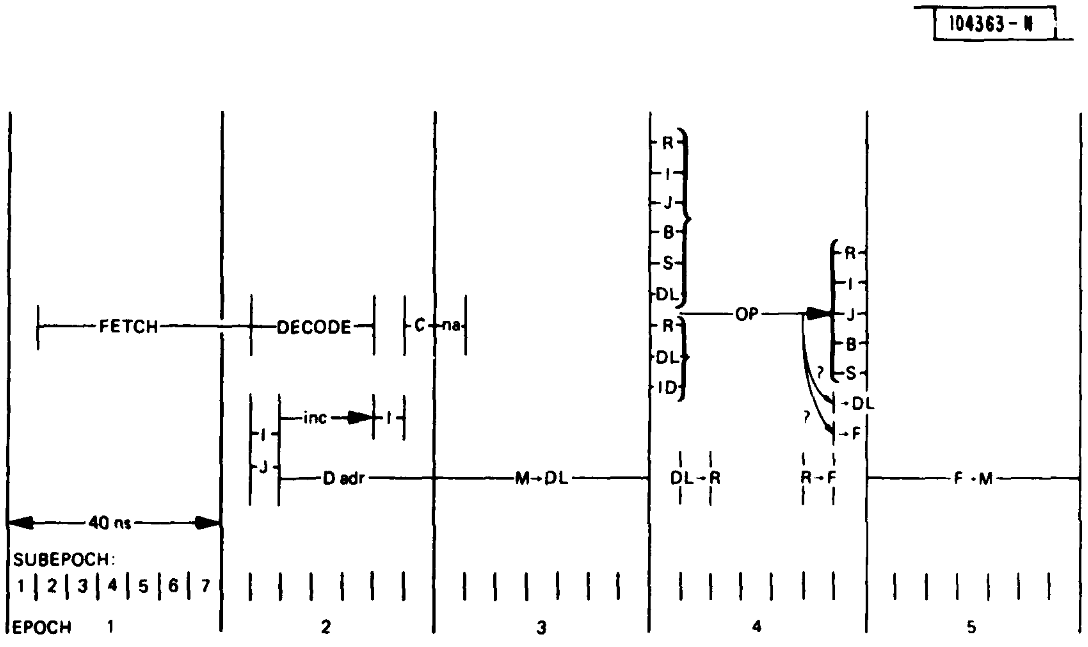
Conceptually, a family of machines has been developed whose three members differ primarily in data memory transfer capabilities. The conceptual differences may be summarized as:

one-transfer: one data memory read or one write;

1.5-transfer: two reads or one write (the option presented in the body of this report);

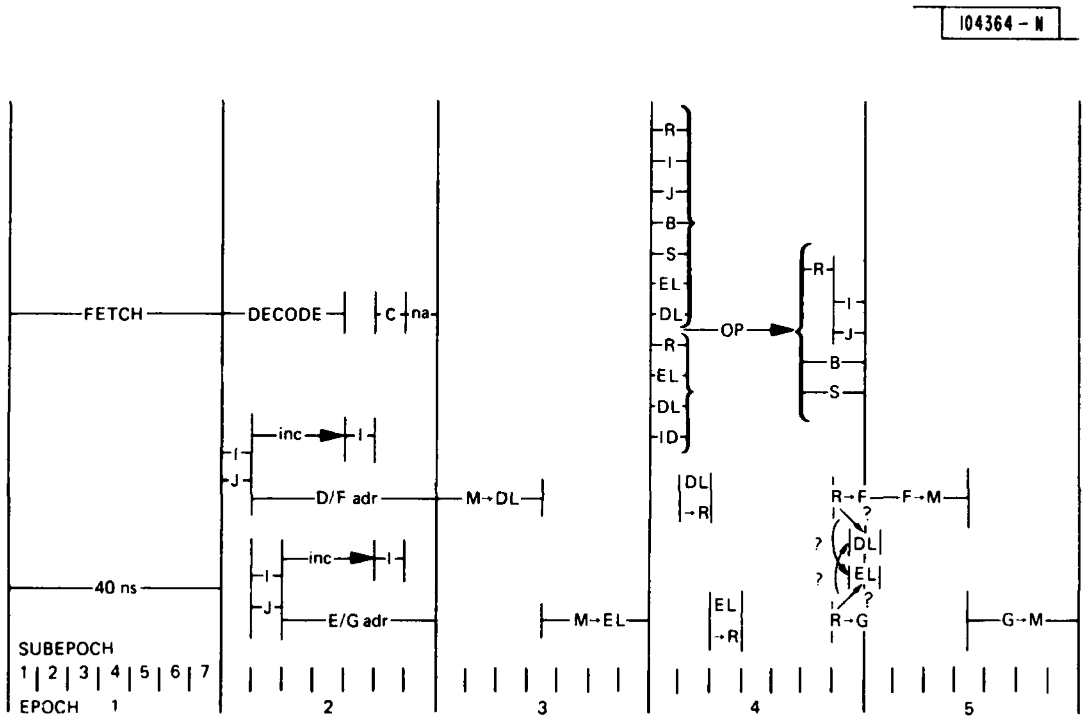
two-transfer: any combination of two data memory transfers (i.e., two reads or two writes or one read and one write.

The instruction pipelines for the one-transfer and two-transfer machines are shown in Figs. A-1 and A-2, respectively. The one-transfer machine can be considered as a stripped version of the 1.5-transfer machine in that only one data memory read address is decoded (epoch 2) and only one data memory read occurs (epoch 3). The consequent relaxed data memory addressing requirements also imply a longer operate time (during epoch 4). The 1.5-transfer machine achieves the single cycle-dual read capability through the use of parallel memories. Data memory writes, though, must occur serially implying that single cycle-dual write of the two-transfer machine can be achieved only through the use of faster RAM. This is illustrated by noting the shortened memory access times appearing in the data memory transfer epochs (3 and 5) of the two-transfer machine instruction pipelines. The main hardware savings of the one-transfer machine over the 1.5-transfer machine are in



SEE FIGURE 4 FOR SYMBOL DEFINITIONS

Fig. A1. Pipeline timing diagram for 1-memory transfer machine. (See Fig. 4 for symbol definitions).



SEE FIGURE 4 FOR SYMBOL DEFINITIONS

Fig. A2. Pipeline timing diagram for 2-memory transfer machine. (See Fig. 4 for symbol definitions).

the data memory, data register and program memory complexes. Only one copy of the data memory array and write queue is needed, saving approximately 54 memory and 39 miscellaneous devices. Also the data register multiplexing and latch logic is simplified, and the program memory word size is reduced by 20 bits due to the elimination of the address field corresponding to the second memory read reference. This saves approximately 20 program memory and 20 support chips. Of equal significance to the raw chip counts is the implied reduction in machine control complexity.

The main additional hardware expense in implementing the two-transfer machine instead of the 1.5-transfer machine stems from the data memory array. To obtain the needed two-memory references in a single machine cycle it is necessary to use the faster but lower scale integration F100422 1K x 1-bit RAM[6] devices (10 ns maximum access time) resulting in an additional 96 memory and 78 distribution chips. In addition, machine control complexity increases significantly over that of either the one- or 1.5-transfer machine.

As shown above, the one- and two-transfer machines, while adhering to the same basic architecture as the 1.5-transfer machine, allow a tradeoff between hardware cost, implementation complexity and software capability. The 1.5-transfer machine was chosen for emphasis in this document because its data transfer capabilities were well matched to the benchmark tasks discussed in Section IV. The improved data transfer rate of the two-transfer machine over the 1.5-transfer machine was not judged worth the considerable increase in complexity in the context of speech signal processing-related problems, although the tradeoff may be favorable in other applications. On the other hand, it was found that the restricted data access capabilities of the one-transfer machine resulted in a notable increase of the programming complexity over the 1.5-transfer variant due to the need to interleave program loop iterations. In summary, the complexity increase of the 1.5-transfer machine over the one-transfer option derives primarily from the addition of parallel hardware while the complexity increase of the two-transfer machine over the 1.5-transfer machine stems from more sophisticated control as well as a considerably larger memory array.

# APPENDIX B

## DIRECT MEMORY ACCESS

A Direct Memory Access (DMA) capability, while not critical to most speech processing tasks, has been found useful in other digital signal processing applications such as image processing and radar. As will be shown below, a DMA capability can be achieved through simple hardware modification of the basic design and integrates well within the existing processor control structure.

During each machine cycle the data memory control evaluates the read and/or write requests and the status of the write queue and takes appropriate action. It has been indicated that no memory transfers occur in a given cycle when the instruction sequence implies neither a read nor a write and the write queue is empty. The low priority DMA scheme presented here detects and utilizes such idle cycles for DMA transfers. An important feature of this scheme is that program execution can proceed normally during DMA activity since program references to data memory have priority over DMA data memory references. Top priority can be implicitly given to the DMA port simply by executing a routine which requires no main memory references (e.g., register and ALU operations).

Figure B-1 shows the hardware configuration of the DMA port within the machine structure. It includes the DMA input and output data lines, the address lines and an I/O port for interrupt driven control. The additional data paths needed in the data memory complex to accommodate the DMA port are shown in Fig. B-2 and include expansion of the data

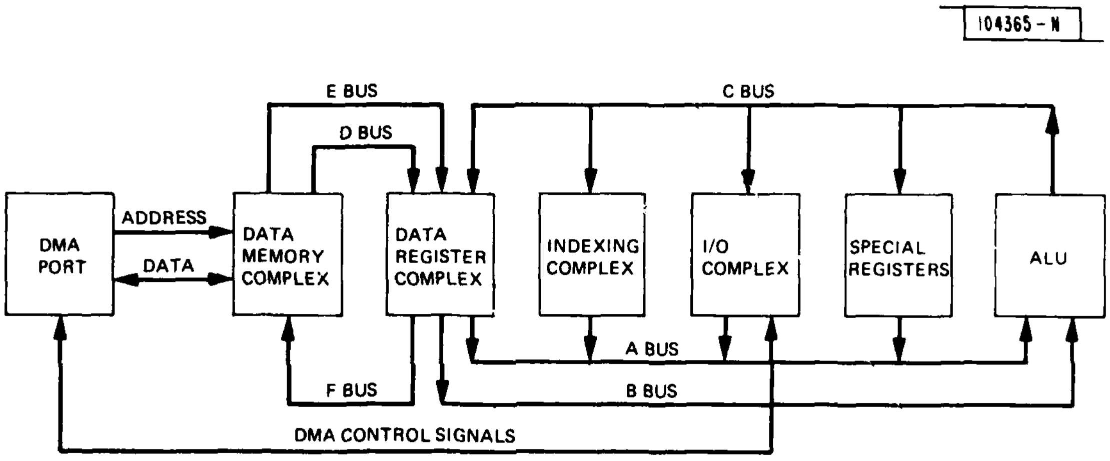


Fig. B1. DMA port interface.

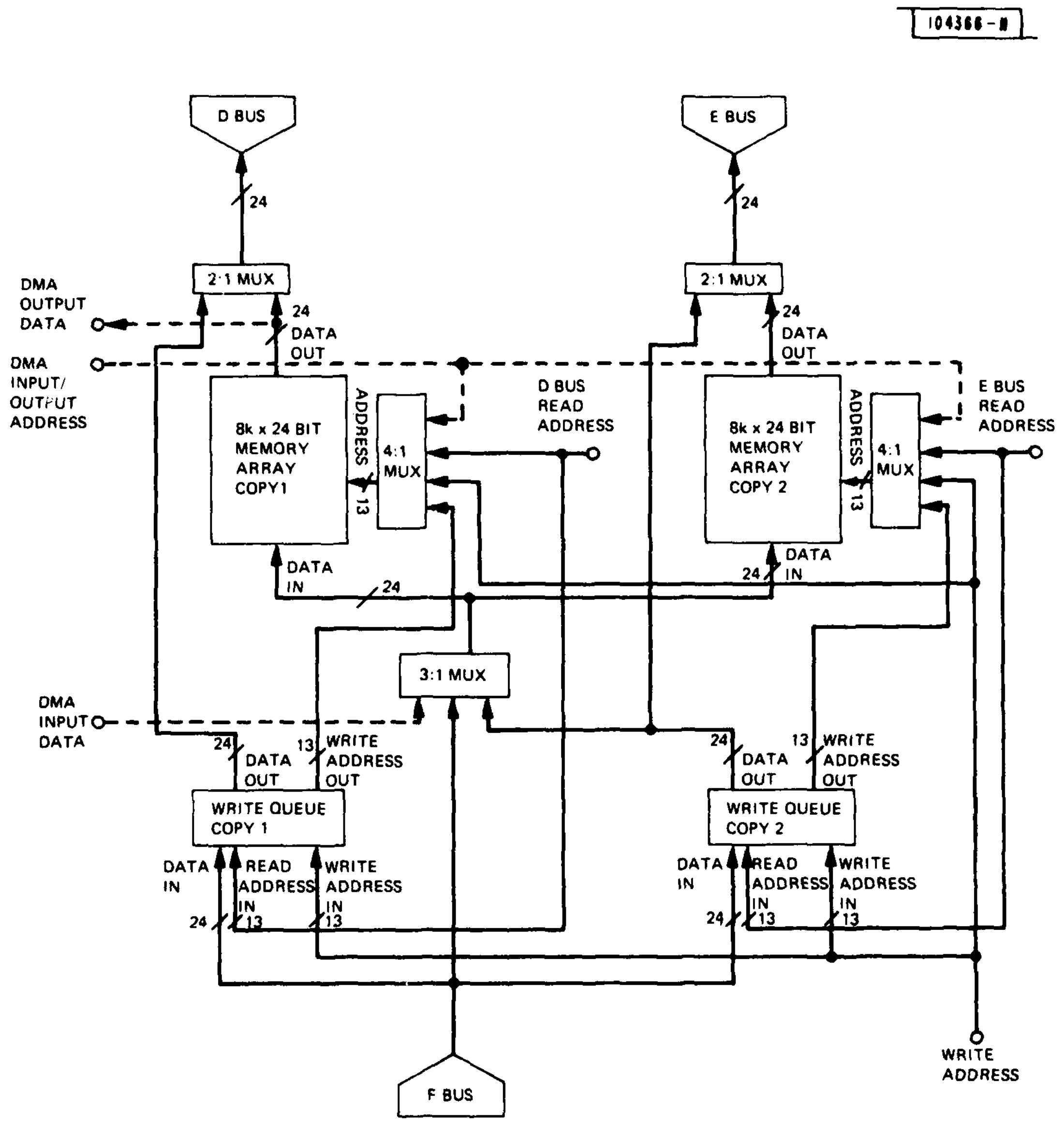


Fig. B2. Expanded data memory system to include DMA.

memory data-in multiplexer to include a DMA input and expansion of the memory array address line multiplexer to include the DMA read and write addresses. The DMA output data may be obtained directly from the memory array data-out lines.

## REFERENCES

1. Bernard Gold, Irwin L. Lebow, Paul G. McHugh and Charles M. Rader, "The FDP, a Fast Programmable Signal Processor," IEEE Trans. on Computers C-20, 1, pp. 33-38 (1971).

2. C. E. Meuhe, P. G. McHugh, W. H. Drury, and B. G. Laird, "The Parallel Microprogrammed Processor (PMP)," Radar 77 Symposium, London, England, 25-28 October 1977.

3. P. E. Blankenship, "LDVT: High Performance Minicomputer for Real-Time Speech Processing," pp. 214a-214g, EASCON '75 Record, Washington, D.C. (29 September - 1 October 1975).

4. The Lincoln Digital Signal Processor is an advanced version of the LDVT. See P. E. Blankenship et al., "The Lincoln Digital Voice Terminal System," Technical Note 1975-53, Lincoln Laboratory, M.I.T. (25 August 1975), DDC AD-A017569/5; or P. E. Blankenship, "LDVT: High Performance Minicomputer for Real-Time Speech Processing," EASCON'75, pp. 214a-214g.

5. ECL Data Book, Fairchild Camera and Instrument Corporation, 464 Ellis Street, Mountain View, CA 94042 (1977).

6. Fairchild data sheets and application notes:
   F100182 9-bit Wallace tree adder
   F100183 2 x 8 bit Recode Multiplier
   F100223 Programmable Interface Unit
   F100402 16 x 4 bit RAM
   F100422 1K x 1 bit RAM
   F100470 4K x 1 bit RAM
   Fairchild Camera and Instrument Corporation, 464 Ellis Street, Mountain View, CA 94042.

7. Barry K. Gilbert, Loren M. Kruger and Rodney D. Beistad, "Design of Prototype Digital Processors Employing Subnanosecond Emitter Coupled Logic and Rapid Fabrication Techniques," IEEE Trans. on Components, Hybrids and Manufacturing Technology CHMT-3, 1, pp. 125-134 (1980).

8. Barry K. Gilbert, personal communication.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER 19 18 ESD-TR-80-217 | 2. GOVT ACCESSION NO. AD-A094 064 | 3. RECIPIENT'S CATALOG NUMBER 9 14 TN-1980-50 |
| 4. TITLE (and Subtitle) 6 A Design Study for an Easily Programmable, High-Speed Processor with a General-Purpose Architecture | | 5. TYPE OF REPORT & PERIOD COVERED Technical Note 9 |
| | | 6. PERFORMING ORG. REPORT NUMBER Technical Note 1980-50 |
| 7. AUTHOR(s) 10 Douglas B. Paul, Vincent J. Sferrino, Joel A. Feldman | | 8. CONTRACT OR GRANT NUMBER(s) 15 F19628-80-C-0002, ARPA Order-3336 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS Lincoln Laboratory, M.I.T. P.O. Box 73 Lexington, MA 02173 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order 3336 Program Element Nos. 61101E/62708E Project Nos. 0D10 and 0T10 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209 | 11 | 12. REPORT DATE 23 Oct[redacted]80 |
| | | 13. NUMBER OF PAGES 78 12 76 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Electronic Systems Division Hanscom AFB Bedford, MA 01731 | | 15. SECURITY CLASS. (of this report) Unclassified |
| | | 15a. DECLASSIFICATION DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

| | | |
|---|---|---|
| signal processor | pipelined control | three-address architecture |
| ECL 100K | general-purpose architecture | register file |

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A design study has been carried out for an easy-to-program high-speed signal processor. The machine achieves a throughput of about 25 million arithmetic operations per second by incorporating parallel addressing and data access into a general-purpose architecture. The study indicates that, for typical signal processing applications, nearly all of the instruction cycles can be used to perform primary arithmetic operations rather than data access or addressing. The proposed machine would require about 1165 ECL 100K chips, occupy 2-3 cubic feet, and consume about 700 watts.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

207650

bpz